Open Trace Format API Specification
Version 1.1

Andreas Knüpfer, Holger Brunst

Center for High Performance Computing
University of Dresden, Germany

{knuepfer|brunst}@zhr.tu-dresden.de

Allen D. Malony, Sameer S. Shende

ParaTools, Inc.

{malony,sameer}@paratools.com

May 18, 2006

Abstract

The Open Trace Format (OTF) is a new trace definition and representation for use with large-scale parallel platforms. OTF addresses three objectives: openness, flexibility, and performance. The OTF specification is provided by this document and represents the first milestone deliverable of the University of California (UC), Lawrence Livermore National Laboratory (LLNL), subcontract #B548849. The second and third phases of the UC/LLNL project will use this specification to implement OTF writing and reading libraries, as part of a complete tracing solution for the LLNL IBM BG/L system.

Contents

1 Introduction

The development of scalable tracing tools for high-performance computing (HPC) platforms with thousands of processors requires both a low-overhead trace measurement system to generate the trace data and efficient trace analysis tools to process the data. Of vital importance to tracing tool development is an open specification of the trace information that provides a target for trace generation and enables trace analysis and visualization tools to operate efficiently at large scale. The integration of facilities for trace generation with trace analysis and visualization tools is facilitated by a well-defined trace format with open, public domain libraries for writing and reading the trace in that format. In addition, features of the trace format can directly support analysis tool capabilities to speedup up trace data access and processing. These two benefits combined addresses concerns for a format that can target future cross-platform tracing solutions for high-end ASC production systems.

The current document gives a detailed specification of the Open Trace Format (OTF), created specifically to support the development of scalable performance tracing tools for the IBM BG/L machine. The specifications covers the trace storage and processing model, record types, and an API specification for reading and writing OTF event traces. The format addresses large applications written in an arbitrary combination of Fortran77, Fortran (90/95/etc.), C, and C++. The trace representation supports efficient scalable access and information processing by structural mechanisms for fast query and features to increase trace processing flexibility.

The OTF specification shall form the basis for a trace measurement and analysis toolset. In particular, the OTF will be demonstrated by its use in the TAU performance system for trace conversion and the Vampir NG (VNG) for trace analysis and visualization. Figure 1 shows the integration of OTF in this suite of tools. TAU presently generates VTF3, EPILOG, and parallel profile files from TAU-formatted traces. The VTF3 files can be input to Vampir/VNG, but OTF is required to enable its full efficiency and performance features. Notice, STF is readable by Vampir/VNG, but lacks the openess of the format.

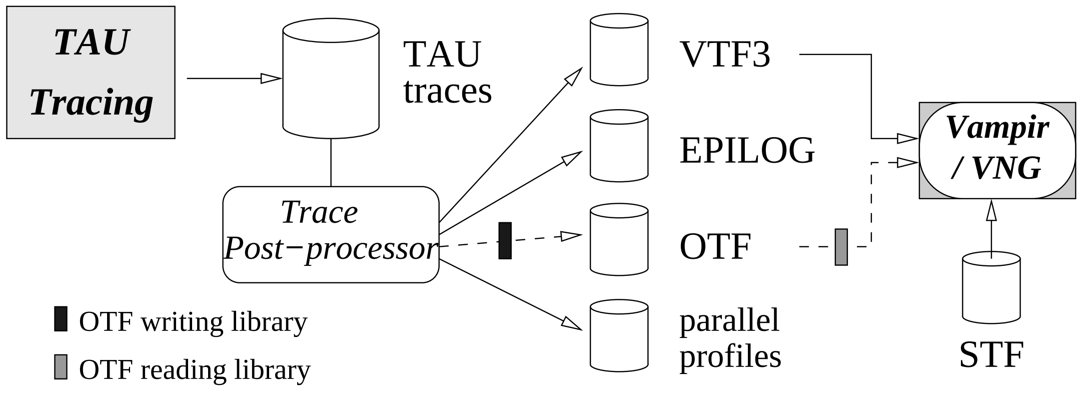

Figure 1: Planned integration of OTF in the TAU Performance Systems and Vampir/VNG.

The remainder of the document is organized in two main sections. Section 2 discusses the design and architecture of OTF. Section 3 presents the application programming interface for OTF, consisting of seven components for reading, writing, managing, and buffering OTF traces. Then the definition of specific records types are listed in Section B. The document concludes with a few OTF application examples in Section 4.

2 OTF Design

The design of OTF is directed at three objectives: openness, flexibility, and performance. The open format defines the record types and file structure so that OTF trace files can be both generated and read correctly. A OTF writing and reading library will be provided later for these purposes. The flexibility objective in the OTF design comes from choices made with regards to trace data representation and storage, as well as parameters that OTF tools can control to organize and work with OTF traces. Performance is determine by how efficient and fast OTF trace query and manipulation can be done. This section discusses the components of the OTF design in the context of these objectives.

2.1 ASCII Format

OTF uses a special ASCII data representation to encode its data items. ASCII encoding allows reduced storage sizes for small values as leading zeros can be omitted. All numbers and tokens are encoded in hexadecimal without the need of a special prefix which allows for a more efficient back and forth transformation compared to decimal numbers. Altogether, this enables a very efficient format with respect to storage size, human readability, and search capabilities on timed event records. Furthermore, it avoids platform dependent byte ordering issues.

2.2 Streams and Files

In order to support fast and selective access to large amounts of performance trace data, OTF is based on a *stream-model*, i. e. single separate units representing segments of the overall data. OTF streams may contain multiple independent processes whereas a process belongs to a single stream exclusively. The latter is needed for consistency reasons and cannot be relaxed without reducing the format's expressiveness.

Each stream is represented by multiple files which store definition records (see Section 2.6), performance events (see Section 2.7), status information (see Section 2.8), and event summaries (see Section 2.9) separately. A single global *master file* holds the necessary information for the process to stream mappings.

The names of trace file parts follow a strict naming convention. Each file name starts with an arbitrary common prefix which can be defined by the user. It is followed by a token identifier used for internal purposes (process mapping) and a suffix according to the file type. OTF files are *not* intended to be accessed directly but through the OTF library's API. This is a strict requirement to guarantee future compatibility.

The master file is always named '`<name>.otf`'. The global definition file is named '`<name>.0.def`'. Events and local definitions are placed in files '`<name>.x.events`' and '`<name>.x.defs`' where the latter files are optional. Snapshots and statistics are placed in files named '`<name>.x.snaps`' and '`<name>.x.stats`' which are optional, too.

When copying, moving or deleting traces it is important to take all according files into account. Deleting or modifying single files of a trace will render the whole trace invalid!

The OTF library allows to transparently read and write trace data independently of the underlying partitioning of streams. Yet, if requested partitioning parameters can be queried and altered. This

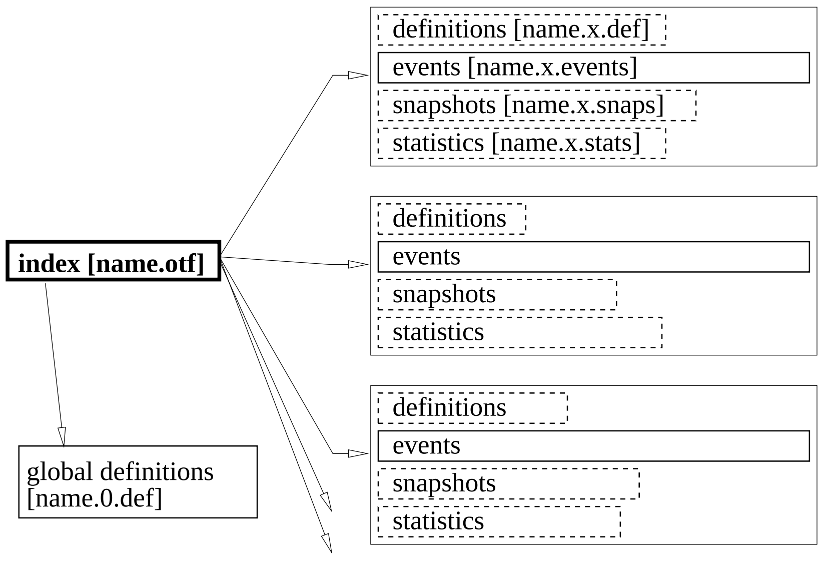

Figure 2: Files belonging to an OTF trace.

additional information can be used for various purposes. A good example is the tuning of the parallel loading process which obviously depends on the inherent process to stream mapping.

Merging with respect to the temporal order of the traced events is done on the fly by the library. Apart from that, the OTF library is able to access trace data consisting of n files by using a given number of m file descriptors with $1 \leq m < n$. This is an important feature to maintain scalability on very complex traces.

Figure 3 gives a high-level view of the OTF architecture showing streams and files used by the trace generation and analysis components.

2.3 State Machine

Within every OTF file, records are arranged as single lines of text whereas the detailed structure of every record type is defined separately.

However, some very frequently used properties are not included in the record lines but are handled by a state machine. This includes time stamp information, process/thread information and maybe others. For those there are special record types internally that set the respective property to a value. This value is then going to be valid for all following records until reassigned. With this approach, for example, time stamps need to be stored only once when multiple records refer to them. The read and write handlers for all record types are not affected by this and will remain as known by existing trace format libraries like VTF3.

2.4 Sorted Streams

Every OTF file needs to be sorted in temporal order. Unsorted files are regarded invalid and no sorting operation will be made available. There is no need to write unsorted traces in the first place.

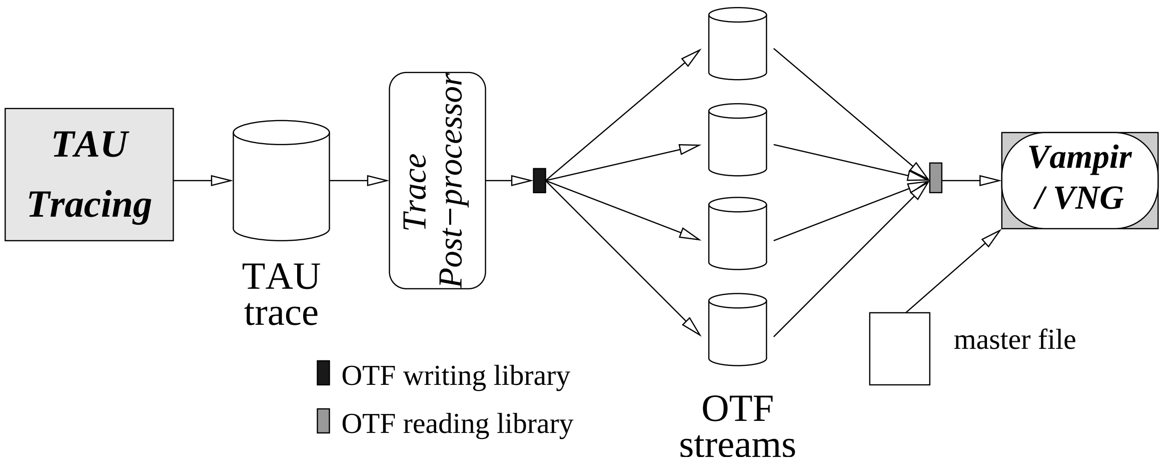

Figure 3: OTF Streams and File Architecture

Furthermore, OTF is designed to handle very huge traces and explicitly sorting streams won't scale well for very big streams.

2.5 Binary Search for Time Stamps

All OTF files can be searched very efficiently for time stamps in order to support fast selective access. As records are always sorted by time stamps such that binary search is applicable. The search mechanism is based on the fact that record boundaries can be quickly identified in the ASCII format.

2.6 Definition Record Types

As usual with trace formats, there is always a number of definition records. Such records carry some global properties like timer resolution, process count, etc. Furthermore, they define tokens to be referenced by event records, which allows for a more efficient encoding.

Definitions can be contained in single streams or globally as desired. There will always be a separate file for definition records. This makes it possible to define tokens late without disturbing the sorting order of events. All definitions are accompanied by a stream identifier which tells the scope of the definition - stream specific or global.

With OTF, all identifiers are tokens, not indices. That means a set of N identifiers is not restricted to $\{0, 1, ..., N-1\}$. The value of zero (0) is always a reserved token used for special purposes. Apart from that, actual numerical values of identifiers are not important. Identifiers are only compared with respect to $=$ resp. $\neq$, where $<$, $>$ and hashing operations might be used for internal optimization of table lookup and so on. Tokens are always of type **uint32_t**, i. e. unsigned integers of 32 bit size.

2.7 Event Record Types

Event records are the actual payload for traces. There is one event file per stream, which is sorted in temporal order.

2.8 Snapshot Record Types

Usually, traces are read linearly from the beginning. As OTF introduces the possibility to access arbitrary time stamps fast, some auxiliary information becomes necessary.

In order to start reading from an arbitrary time stamp, the current state of all participating processes needs to be known. If this information is not available from having read all preceding records as well, it needs to be stored explicitly. This is what *snapshot records* are designed for.

Snapshots provide the call stack (i.e. all active function calls), a list of pending messages, ongoing I/O activities, current OpenMP regions, etc... at a point in time (**not** including events at that very time stamp itself). Based on this information one can start reading event records at that very time stamp.

Snapshots are not generated by the OTF library itself but must explicitly be added. However, because they live in a separate fi le, it is possible to add/manipulate/replace/delete snapshots of a stream without affecting event data.

It is suggested to create snapshot information for time stamps on regular time distances. Different granularities for different phases of a trace might be convenient as well. Snapshots can be added right after trace fi le generation as an automated batch job or later on based on specifi c preferences.

2.9 Statistical Summary Record Types

A second class of auxiliary information is provided by *summary records*. They provide an overview over a whole interval of time, which might serve as a hint whether to read all events of that interval of time or not.

The data provided for this purpose is not explicit values for particular intervals of time but in a differential fashion like follows:

In order to provide summary information about a monotonous increasing property $p(t)$ for a time interval $[a, b]$, store the values $p(a)$ and $p(b)$. The result $p([a, b])$ can be computed as

$$p([a, b]) := p(b) - p(a).$$

With n points in time $t_0, ...t_{n-1}$, there are $n * (n - 1)$ possible interval results $p([t_i, t_j]), i \neq j$ of varying granularity that can be queried directly.

In comparison, with n explicit intervals potentially expensive accumulation of multiple (small) time intervals would be necessary to query for more than the n basic intervals.

Like snapshots, summaries can be added/modifi ed/replaced/deleted without affecting events. Also, they need to be created explicitly and are not generated by the OTF library.

3 Application Programming Interface

The application programming interface (API) consists of seven components as shown in Figure 4. There are high level trace read and write interfaces called `Reader` and `Writer` that address whole traces. Both refer to the stream management interface `Master`. For dealing with single streams, read and write access is handled by `RStream` and `WStream`. Finally, the low level access to single fi les is left to the `RBuffer` resp. `WBuffer`. The five high level interfaces are intended to be public, while the two low level interfaces are for internal use only.

All seven components are described below. See Section 4 for typical application examples for them. All OTF components and interfaces are in pure C.

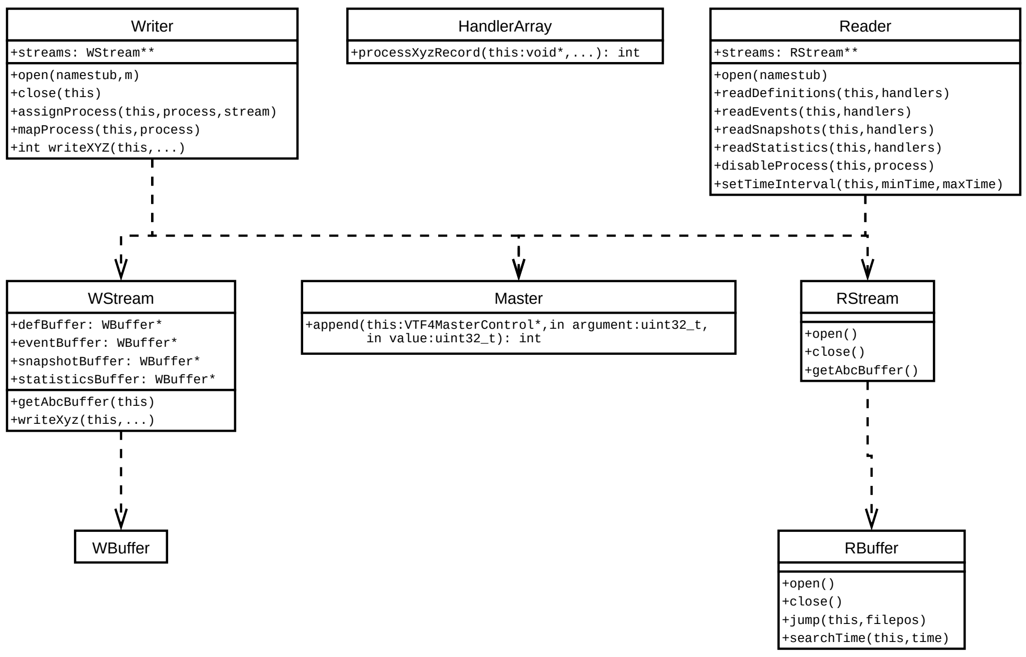
Writer
+streams: WStream**
+open(namestub,m)
+close(this)
+assignProcess(this,process,stream)
+mapProcess(this,process)
+int writeXYZ(this,...)
HandlerArray
+processXyzRecord(this:void*,...): int
Reader
+streams: RStream**
+open(namestub)
+readDefinitions(this,handlers)
+readEvents(this,handlers)
+readSnapshots(this,handlers)
+readStatistics(this,handlers)
+disableProcess(this,process)
+setTimeInterval(this,minTime,maxTime)
WStream
+defBuffer: WBuffer*
+eventBuffer: WBuffer*
+snapshotBuffer: WBuffer*
+statisticsBuffer: WBuffer*
+getAbcBuffer(this)
+writeXyz(this,...)
Master
+append(this:VTF4MasterControl*,in argument:uint32_t, in value:uint32_t): int
RStream
+open()
+close()
+getAbcBuffer()
WBuffer
RBuffer
+open()
+close()
+jump(this,filepos)
+searchTime(this,time)

Figure 4: OTF API Classes Overview

3.1 Trace Write Interface

OTF has a low level and a high level trace writing interface. Latter is for simultaneous writing of multiple streams (files). The low level interface targets a single streams only.

3.1.1 Global Write Interface

The class **OTF_Writer** and the associated functions are for writing traces with multiple streams.

OTF_Writer* `OTF_Writer_open(` **char*** `fileNamePrefix,` **uint32_t** `numberOfStreams,` **OTF_FileManager*** `fileManager );`

Open a new OTF_Writer with `numberOfStreams` automatic streams, `numberOfStreams=0` means unlimited. Processes are assigned to up to `numberOfStreams` streams on demand. If processes are assigned with a call to `OTF_Writer_assignProcess()` explicitly after `numberOfStreams` streams are present already, this limit could be exceeded.

`int OTF_Writer_close(` **OTF_Writer*** `writer );`

Close all files and delete the OTF_Writer object.

void `OTF_Writer_setFormat(` **OTF_Writer*** `writer,`
uint32_t `format );`

Set the default ouput format. The format is applied to all streams opened by the writer. `format` may be `OTF_WSTREAM_FORMAT_SHORT` or `OTF_WSTREAM_FORMAT_LONG`.

int `OTF_Writer_setCompression(` **OTF_Writer*** `writer,`
OTF_FileCompression `compression );`

Set the standard compression method for all buffers managed by this writer. `compression` may be `OTF_FILECOMPRESSION_UNCOMPRESSED` or `OTF_FILECOMPRESSION_COMPRESSED`. The function returns 1 on success or 0 on error.

uint32_t `OTF_Writer_assignProcess(` **OTF_Writer*** `writer,`
uint32_t `process,` **uint32_t** `stream );`

Assign the given ’process’ to the specifi ed ’stream’ explicitly. Mind that 0 is not a valid stream id but a reserved value. Return error code, where 0 marks success.

int `OTF_Writer_write<RecordType>(` **OTF_Writer*** `writer, ...);`

Writes a record of type <Record Type> to an open stream. There is a specifi c version for every record type with customized signature. See Appendix A for a complete description. Every record is written to the proper stream and to the appropriate fi le inside each stream.

3.1.2 Local Write Interface

The class **OTF_WStream** and the associated functions are for writing single streams of a trace.

OTF_WStream* `OTF_WStream_open(` **const char*** `namestub,`
uint32_t `id,`
OTF_FileManager* `fileManager );`

Open a new writer stream with name prefi x ’namestub’ and token ’id’ which must be unique.

int `OTF_WStream_close(` **OTF_WStream*** `wstream );`

Close and delete an open **OTF_WStream** object.

OTF_WBuffer* `OTF_WStream_getDefBuffer(`
OTF_WStream* `wstream );`

Return the current streams defi nitions buffer, which is allocated on demand if not already existing.

OTF_WBuffer* `OTF_WStream_getEventBuffer(`
OTF_WStream* `wstream );`

Return the current streams events buffer, which is allocated on demand if not already existing.

OTF_WBuffer* `OTF_WStream_getSnapshotBuffer(`
OTF_WStream* `wstream );`

Return the current streams snapshots buffer, which is allocated on demand if not already existing.

OTF_WBuffer* `OTF_WStream_getStatsBuffer(`
OTF_WStream* `wstream );`

Return the current streams summaries buffer, which is allocated on demand if not already existing.

int `OTF_WStream_write<Record Type>(` **OTF_WStream*** `wstream, ...);`

Write a record of type <Record Type> to the stream 'wstream'. Every record is written to the appropriate file inside the current stream.

3.1.3 Low Level File Write Interface

The class **OTF_WBuffer** and the associated functions are for writing single files of a stream. This is for internal use only.

OTF_WBuffer* `OTF_WBuffer_open(` **const char*** `filename,`
OTF_FileManager* `fileManager );`

constructor - internal use only

int `OTF_WBuffer_close(` **OTF_WBuffer*** `wbuffer );`

destructor - internal use only

int `OTF_WBuffer_setSize(` **OTF_WBuffer*** `wbuffer,` **size_t** `size );`

Set the size of the memory buffer. Cannot shrink buffer but only extend afterwards.

int `OTF_WBuffer_flush(` **OTF_WBuffer*** `wbuffer );`

Writes the buffer contents to file and marks the buffer empty again.

int `OTF_WBuffer_guarantee(` **OTF_WBuffer*** `wbuffer,`
size_t `space );`

Ask the buffer to guarantee at least 'space' bytes at current writing position before the next flush is necessary. Return 1 on success.

int `OTF_WBuffer_setTimeAndProcess(` **OTF_WBuffer*** `wbuffer,`
uint64_t `t,` **uint32_t** `p );`

Set process state machine to 'p' and time stamp state machine to 't'. If 'p' is the current process and 't' is the current time stamp, nothing is done. If the process has changed, a process record will be written. If the time has changed, the new time stamp and the current process will be written. If ’t’ is lower than the current time stamp, it is regarded as an error. Return != 1 on success and 0 on error.

Furthermore, there are basic write operations modifying the memory buffer:

uint32_t `OTF_WBuffer_writeKeyword(` **OTF_WBuffer*** `wbuffer,`
const char* `keyword );`

Append a keyword to the write buffer. A key word is a string without quotes. Buffer flush is done if necessary. Return the number of bytes written.

uint32_t `OTF_WBuffer_writeString(` **OTF_WBuffer*** `wbuffer,`
const char* `string );`

Append a string to the write buffer. A string is surrounded by quotes. Buffer flush is done if necessary. Return the number of bytes written.

uint32_t `OTF_WBuffer_writeChar(` **OTF_WBuffer*** `wbuffer,`
const char `character );`

Append a char to the write buffer. Buffer flush is done if necessary. Return the number of bytes written (=1).

uint32_t `OTF_WBuffer_writeUint32(` **OTF_WBuffer*** `wbuffer,`
uint32_t `value );`

This function appends an unsigned integer ’value’ in hex format to the write buffer. Buffer flush is done if necessary. The return value is the number of characters written.

uint32_t `OTF_WBuffer_writeUint64(` **OTF_WBuffer*** `wbuffer,`
uint64_t `value );`

This function appends an 64bit unsigned integer ’value’ in hex format to the write buffer. Buffer flush is done if necessary. The return value is the number of characters written.

uint32_t `OTF_WBuffer_writeNewline(` **OTF_WBuffer*** `wbuffer );`

Append a newline character to the buffer. Buffer flush is done if necessary. Return the number of bytes written.

3.2 Trace Read Interface

Similar to the writing interface, OTF comes with a dual layer reading interface. The global interface provides transparent access to multiple streams while the local interface allows access to single streams only.

3.2.1 Global Read Interface

The class **OTF_Reader** and the associated functions are for reading traces with multiple streams.

OTF_Reader* `OTF_Reader_open(` **const char*** `namestub,`
OTF_FileManager* `fileManager );`

Open OTF trace.

int `OTF_Reader_close(` **OTF_Reader*** `reader );`

Close OTF trace and delete the **OTF_Reader*** object.

int `OTF_Reader_readDefinitions(` **OTF_Reader*** `reader,`
OTF_HandlerArray* `handlers );`

This function reads definition records from trace and calls the appropriate handlers from the **OTF_HandlerArray** object given. The default value **count**=0 will read all available records.

Inside the call-back, handlers may return several pre-defined constants in order to influence he bahaviour of OTF, like **OTF_RETURN_NULL**, **OTF_RETURN_BREAK**, **OTF_RETURN_ABORT**.

int `OTF_Reader_readEvents(` **OTF_Reader*** `reader,`
OTF_HandlerArray* `handlers );`

This function reads event records from trace and calls the appropriate handlers from the **OTF_HandlerArray** object given.

Compare **OTF_Reader_readDefinitions()**.

int `OTF_Reader_readSnapshots(` **OTF_Reader*** `reader,`
OTF_HandlerArray* `handlers );`

This function reads snapshot records from trace and calls the appropriate handlers from the **OTF_HandlerArray** object given.

Compare **OTF_Reader_readDefinitions()**.

int `OTF_Reader_readStatistics(` **OTF_Reader*** `reader,`
OTF_HandlerArray* `handlers );`

This function reads summary records from trace and calls the appropriate handlers from the **OTF_HandlerArray** object given.

Compare **OTF_Reader_readDefinitions()**.

OTF_RStream* `OTF_Reader_getStream(` **OTF_Reader*** `reader,`
uint32_t `id );`

Search the stream with the given ’id’ and return it.

int `OTF_Reader_disableProcess(` **OTF_Reader*** `reader,`
uint32_t `processId );`

Remove the process with the given 'processid' from active process. That means filtering out this process from reading - no records associated with this process are delivered anymore. Return 1 on success.

`void OTF_Reader_setTimeInterval(` **OTF_Reader*** `reader,`
uint64_t `minTime,` **uint64_t** `maxTime );`

Set 'minTime' and 'maxTime' of OTF_Reader. That means further reading is restricted resp. filtered according to this time interval.

void `OTF_Reader_reset(` **OTF_Reader*** `reader );`

Reset all processes active and the time interval to the default $[0, \infty]$.

3.2.2 Local Read Interface

The class **OTF_RStream** and the associated functions are for reading single streams of a trace.

OTF_RStream* `OTF_RStream_open(` **const char*** `namestub,`
uint32_t `id,`
OTF_FileManager* `fileManager );`

Open an existing stream with 'id'.

int `OTF_RStream_close(` **OTF_RStream*** `rstream );`

Close and delete an open stream reader object.

int `OTF_RStream_readDefinitions(` **OTF_RStream*** `rstream,`
OTF_HandlerArray* `handlers );`

This function reads definition records from stream and calls the appropriate handlers from the **OTF_HandlerArray** object given. The default value **count**=0 will read all available records.

Inside the call-back, handlers may return several pre-defined constants in order to influence he bahaviour of OTF, like **OTF_RETURN_NULL**, **OTF_RETURN_BREAK**, **OTF_RETURN_ABORT**.

int `OTF_RStream_readEvents(` **OTF_RStream*** `rstream,`
OTF_HandlerArray* `handlers );`

This function reads event records from stream and calls the appropriate handlers from the **OTF_HandlerArray** object given.

Compare **OTF_RStream_readDefinitions()**.

int `OTF_RStream_readSnapshots(` **OTF_RStream*** `rstream,`
OTF_HandlerArray* `handlers );`

This function reads snapshot records from stream and calls the appropriate handlers from the **OTF_HandlerArray** object given.

Compare **OTF_RStream_readDefinitions()**.

int `OTF_RStream_readStatistics(` **OTF_RStream*** `rstream,`
OTF_HandlerArray* `handlers );`

This function reads summary records from stream and calls the appropriate handlers from the **OTF_HandlerArray** object given.

Compare **OTF_RStream_readDefinitions()**.

OTF_RBuffer* `OTF_RStream_getDefBuffer(`
OTF_RStream* `rstream );`

Return the streams defi nition buffer, opened on demand.

OTF_RBuffer* `OTF_RStream_getEventBuffer(`
OTF_RStream* `rstream );`

Return the streams event buffer, opened on demand.

OTF_RBuffer* `OTF_RStream_getSnapsBuffer(`
OTF_RStream* `rstream );`

Return the streams snapshots buffer, opened on demand.

OTF_RBuffer* `OTF_RStream_getStatsBuffer(`
OTF_RStream* `rstream );`

Return the streams summary buffer, opened on demand.

3.2.3 Low Level File Read Interface

The class **OTF_RBuffer** and the associated functions are for reading single fi les of a stream. This is for internal use only.

OTF_RBuffer* `OTF_RBuffer_open(` **const char*** `filename,`
OTF_FileManager* `fileManager );`

constructor - internal use only

int `OTF_RBuffer_close(` **OTF_RBuffer*** `rbuffer );`

destructor - internal use only

int `OTF_RBuffer_setSize(` **OTF_RBuffer*** `rbuffer,` **size_t** `size );`

Set memory buffer size. Cannot shrink buffer but only extend.

int `OTF_RBuffer_setJumpSize(` **OTF_RBuffer*** `rbuffer,`
size_t `size );`

Set 'jumpsize', a parameter for binary searchin a file. Return 0 if 'size' is greater than the buffer size.

char* `OTF_RBuffer_getRecord(` **OTF_RBuffer*** `rbuffer );`

Make the next record availabe from the buffer. Return the pointer to the record string which is terminated by '\n' not '\0' ! This function must be called before any record access. It ensures the record is available completely in the buffer. Furthermore, time and process information is kept track of. It is recommended to use the 'OTF_RBuffer_readXXX()' functions below to read record components instead of parsing manually. In any case, after reading 'OTF_RBuffer_readNewline()' needs to be called which proceeds to the next record begin no matter if there are still characters from the current record present or not.

int `OTF_RBuffer_guaranteeRecord(` **OTF_RBuffer*** `rbuffer );`

Ask the buffer to guarantee at least one complete record at the current read position inside the buffer. This means one line, e.g. '\n' character. If no complete record is found the buffer has to be advanced by reading new contents from file. Return 1 on success, 0 means the file is exceeded.

`void OTF_RBuffer_printRecord(` **OTF_RBuffer*** `rbuffer );`

Print the record at the current buffer position, i.e. until the next newline character. This is for debugging purposes only and won't modify the buffer in any way.

int `OTF_RBuffer_jump(` **OTF_RBuffer*** `rbuffer,` **uint64_t** `filepos );`

Jump to the given file position and restore buffer and references as if the buffer had reached the position by advancing through the file linearly. In particular, find the next record start, then find next time stamp and process specification in order to set 'time' and 'process' to true values. Return error code 1 on success. Otherwise, the file is not that large or there are no appropriate time and process specifications on the tail of the file. In that case the buffer contents is undefined.

uint64_t `OTF_RBuffer_readUint64(` **OTF_RBuffer*** `rbuffer );`

Read an 64bit unsigned integer in hex format from buffer and return it.

uint32_t `OTF_RBuffer_readUint32(` **OTF_RBuffer*** `rbuffer );`

Read an unsigned integer in hex format from buffer and return it.

const char* `OTF_RBuffer_readString(` **OTF_RBuffer*** `rbuffer );`

Read a string from buffer and return it.

uint32_t `OTF_RBuffer_readArray(` **OTF_RBuffer*** `rbuffer,`
uint32_t* `array );`

Read a array of **uint32_t** integers from buffer and return the number of elements.

int `OTF_RBuffer_testChar(` **OTF_RBuffer*** `rbuffer,` **char** `c );`

Test if the next character equals the given one (leading spaces are ignored). If the right character is found, return 1, and advance by 1 step. If the character was not found, keep the buffer position such that the test can be repeated for another character.

int `OTF_RBuffer_testString(` **OTF_RBuffer*** `rbuffer,`
const char* `string );`

Test if the next string equals the given one (leading spaces are ignored). If the right string is found, return 1, and advance the buffer position. If the string was not found, keep the buffer position such that the test can be repeated for another string.

int `OTF_RBuffer_readNewline(` **OTF_RBuffer*** `rbuffer );`

Read a newline such that the buffer position is at the next record beginning. Skip all characters found, assume they are to be ignored. Return 1 on success, 0 on error.

`void OTF_RBuffer_skipSpaces(` **OTF_RBuffer*** `rbuffer );`

Advance the buffer position while there are spaces.

`void OTF_RBuffer_skipKeyword(` **OTF_RBuffer*** `rbuffer );`

Advance the buffer position while there are capital letters.

uint64_t `OTF_RBuffer_getCurrentTime(` **OTF_RBuffer*** `rbuffer );`

Return the current time of the buffer from state machine.

`void OTF_RBuffer_setCurrentTime(` **OTF_RBuffer*** `rbuffer,`
uint64_t `time );`

Set the current time of the buffers state machine to the given one.

uint32_t `OTF_RBuffer_getCurrentProcess(`
OTF_RBuffer* `rbuffer );`

Return the current process of the buffer from state machine.

`void OTF_RBuffer_setCurrentProcess(` **OTF_RBuffer*** `rbuffer,`
uint32_t `process );`

Set the current process of the buffers state machine to the given one.

int `OTF_RBuffer_searchTime(` **OTF_RBuffer*** `rbuffer,`
uint64_t `time );`

Search the buffer for the given time and set the buffer position to the next record after that time. Return 1 on success, 0 on error.

int `OTF_RBuffer_getFirstLastTime(` **OTF_RBuffer*** `rbuffer,`
uint64_t `filesize );`

Determine buffers firstTime and lastTime if not already set. Return 1 on success, 0 on error.

3.3 Stream Management Interface

The class **OTF_MasterControl** handles the mapping from processes to streams and vice versa. It is intended for internal use only as its functionality can be accessed by higher level interfaces. This is the basic interface (not complete):

OTF_MasterControl* `OTF_MasterControl_new();`

Create an empty **OTF_MasterControl** structure. The returned object must be freed by **OTF_MasterControl_finish()**.

OTF_MasterControl* `OTF_MasterControl_read(`
const char* `namestub );`

Read a master control file according to namestub and initialize the newly created **OTF_MasterControl** structure accordingly. The returned object must be freed by **OTF_MasterControl_finish()**.

int `OTF_MasterControl_finish(` **OTF_MasterControl*** `masterCtrl );`

Destructor, delete **OTF_MasterControl** object.

int `OTF_MasterControl_write(` **OTF_MasterControl*** `masterCtrl,`
const char* `namestub );`

Write a master control file with the current contents of the given object, return 1 on success.

int `OTF_MasterControl_append(` **OTF_MasterControl*** `masterCtrl,`
uint32_t `argument,` **uint32_t** `value );`

Append the mapping *argument to value* to the master control structure, return 0 if this conflicts with the current mapping.

uint32_t `OTF_MasterControl_getNewStreamId(`
OTF_MasterControl* `masterCtrl );`

Return a previously unused argument. Of course, one cannot avoid collisions with arguments explicitly defined later on.

OTF_MapEntry* `OTF_MasterControl_getEntry(`
OTF_MasterControl* `masterCtrl,` **uint32_t** `argument );`

Return entry for the given argument or NULL.

uint32_t `OTF_MasterControl_mapReverse(` **OTF_MasterControl*** `masterCtrl,`
uint32_t `value );`

Return the argument to the given value. If no mapping was defined make up a new one.

int `OTF_MasterControl_check(` **OTF_MasterControl*** `masterCtrl );`

Check if the current mapping is consistent in itself. Return 1 on success.

3.4 Handler Class

The class **OTF_HandlerArray** contains a list of record type specific handlers with customized signatures.

OTF_HandlerArray* `OTF_HandlerArray_open();`

Open a new array of handlers.

int `OTF_HandlerArray_close(` **OTF_HandlerArray*** `handlers );`

Close and delete a OTF_HandlerArray object.

int `OTF_HandlerArray_setHandler(` **OTF_HandlerArray*** `handlers,`
OTFFunctionPointer* `pointer,` **uint32_t** `recordtype );`

Assign the function pointer to your own handler of the given record type. Appendix B lists all supported record types and their respective handler parameterizations.

All call-back handlers have an `int` return value which is supposed to deliver 0. Other return values are regarded as an error and reading will terminate immediately.

int `OTF_HandlerArray_getCopyHandler(` **OTF_HandlerArray*** `handlers,`
OTF_Writer* `writer );`

Provide copy handlers directly to all record types that use the given **OTF_Writer*** object for output.

int `OTF_HandlerArray_setFirstHandlerArg(`
OTF_HandlerArray* `handlers,` **void*** `firsthandlerarg,`
uint32_t `recordtype );`

Assign the *first argument* to your own handler of the given record type. The *first argument* is an arbitrary pointer of type **void***. This pointer provides a user data structure inside the handler call without the necessity of using a global variable. The user supplied data structure has to be casted to **void*** forth and back in order to comply with generic interface.

4 Application Examples

Some typical application examples shall emphasize the usage of the different interfaces.

4.1 Trace Library

Within a trace library, the OTF library will be used in order to flush in-memory event buffers to trace file once in a while. Usually this is performed by every process/thread independently. Communication inside the trace library should be avoided if possible.

Under those conditions, the local write interface is most convenient. Every process/thread will open an exclusive stream **OTF_WStream** with an unique identifier. Thus, all N processes can write their events to N different stream independently.

Definition records can be added at any time: in the beginning, during tracing or in the end. If there are local (process/thread specific) definitions, they go the the respective stream. Global definitions can be written to the special stream with identifier 0, which is reserved for such purposes.

Finally, one process (the master) has to write the master control file, which states the mapping of processes/threads to streams. In this case, it will be a simple $1:1$ mapping. So, there is a valid trace without merging or reprocessing. The event data need not to be touched again.

However, reprocessing might be necessary in order to synchronize timers, translate different local token sets to a single consistent global token set or other post-processing steps. Furthermore, snapshot and summary information are to be added to this trace later. This might be combined with explicit merging like explained in the next scenario.

4.2 Trace Merging

Merging is not strictly necessary if traces are generated like outlined above. However, one might want to create a different granularity with the mapping of processes/thread to streams. For example, one wants to have fixed number of $M < N$ streams such that efficient parallel input is possible on a parallel analysis environment with M nodes. Or a fixed number of K processes/threads might be desirable.

The actual merging operation is supported by the OTF library itself already. Everything that is left to do is reading a trace with the given granularity and writing it again with another. Therefore, reading should be performed via the global read interface. Otherwise, merging is not handled by OTF.

The copy handlers provided by the read interface can be used directly. It requires just an **OTF_Writer** object of the global write interface. It has to be initialized with the number of output streams. Alternatively, the mapping can also be defined explicitly.

If, besides merging, additional translation or filtering is to be performed, customized handlers that receive the records, manipulate them and pass them on to a **OTF_Writer** object, should be used.

It is possible to do merging in parallel if the input streams are distributed in a disjoint way.

4.3 Parallel Reading and Searching

Reading a trace for analysis provides a number of new possibilities with OTF. Of course, classical linear reading from beginning to end is supported.

First, every analysis application will read definitions - global as well as all local ones. This is supposed to be done by every process of an parallel/distributed analysis application. All further reading can take advantage of efficient parallel input. Every analysis process can select only a subset of the trace processes to read. For best efficiency, this mapping should be aligned to the existing processes to streams mapping such that every reader process accesses a minimum number of different streams.

Second, it might refer to summary records (provided the trace contains some) in order to find interesting spots for selective access.

Third, the application might decide to read a selected time interval $\mathbb{I} = t_0, t_1$ only. Then it has to search for the latest snapshot time stamp $t^* < \mathbb{I}$. This search is performed by OTF via binary search which is very fast and efficient.

Processing that snapshot enables the analysis application to start reading from time t^* in the event time stamp. This time stamp inside the event stream is again detected by OTF via binary search. Now, event by event is delivered by the callback handlers as usual. Multiple select and search operations can be performed subsequently or in parallel.

5 Conclusion

This document represents the first milestone deliverable of the University of California (UC), Lawrence Livermore National Laboratory (LLNL), subcontract #B548849. The OTF specification reported here will be updated during the course of the UC/LLNL project as necessary to reflect decisions made in the second and third implementation phases of the project. No substantial changes to the specification are anticipated.

A Global Trace Record Output Handler Interface

Functions

- **OTF_Writer** ∗ **OTF_Writer_open** (char ∗fi leNamePrefi x, uint32_t numberOfStreams, OTF_-FileManager ∗fi leManager)
- int **OTF_Writer_close** (**OTF_Writer** ∗writer)
- int **OTF_Writer_closeAllStreams** (**OTF_Writer** ∗writer)
- int **OTF_Writer_setCompression** (**OTF_Writer** ∗writer, OTF_FileCompression compression)
- OTF_FileCompression **OTF_Writer_getCompression** (**OTF_Writer** ∗writer)
- void **OTF_Writer_setBufferSizes** (**OTF_Writer** ∗writer, uint32_t size)
- uint32_t **OTF_Writer_getBufferSizes** (**OTF_Writer** ∗writer)
- void **OTF_Writer_setFormat** (**OTF_Writer** ∗writer, uint32_t format)
- uint32_t **OTF_Writer_getFormat** (**OTF_Writer** ∗writer)
- uint32_t **OTF_Writer_assignProcess** (**OTF_Writer** ∗writer, uint32_t process, uint32_-t stream)
- OTF_MasterControl ∗ **OTF_Writer_getMasterControl** (**OTF_Writer** ∗writer)
- void **OTF_Writer_setMasterControl** (**OTF_Writer** ∗writer, OTF_MasterControl ∗mc)
- int **OTF_Writer_writeDefinitionComment** (**OTF_Writer** ∗writer, uint32_t stream, const char ∗comment)
- int **OTF_Writer_writeDefTimerResolution** (**OTF_Writer** ∗writer, uint32_t stream, uint64_t ticksPerSecond)
- int **OTF_Writer_writeDefProcess** (**OTF_Writer** ∗writer, uint32_t stream, uint32_t process, const char ∗name, uint32_t parent)
- int **OTF_Writer_writeDefProcessGroup** (**OTF_Writer** ∗writer, uint32_t stream, uint32_-t procGroup, const char ∗name, uint32_t numberOfProcs, const uint32_t ∗procs)
- int **OTF_Writer_writeDefFunction** (**OTF_Writer** ∗writer, uint32_t stream, uint32_t func, const char ∗name, uint32_t funcGroup, uint32_t source)
- int **OTF_Writer_writeDefFunctionGroup** (**OTF_Writer** ∗writer, uint32_t stream, uint32_t funcGroup, const char ∗name)
- int **OTF_Writer_writeDefCollectiveOperation** (**OTF_Writer** ∗writer, uint32_t stream, uint32_t collOp, const char ∗name, uint32_t type)
- int **OTF_Writer_writeDefCounter** (**OTF_Writer** ∗writer, uint32_t stream, uint32_t counter, const char ∗name, uint32_t properties, uint32_t counterGroup, const char ∗unit)
- int **OTF_Writer_writeDefCounterGroup** (**OTF_Writer** ∗writer, uint32_t stream, uint32_t counterGroup, const char ∗name)
- int **OTF_Writer_writeDefScl** (**OTF_Writer** ∗writer, uint32_t stream, uint32_t source, uint32_t sourceFile, uint32_t line)
- int **OTF_Writer_writeDefSclFile** (**OTF_Writer** ∗writer, uint32_t stream, uint32_t sourceFile, const char ∗name)
- int **OTF_Writer_writeOtfVersion** (**OTF_Writer** ∗writer, uint32_t stream)
- int **OTF_Writer_writeDefCreator** (**OTF_Writer** ∗writer, uint32_t stream, const char ∗creator)

- int **OTF_Writer_writeEnter** (**OTF_Writer** ∗writer, uint64_t time, uint32_t function, uint32_t process, uint32_t source)
- int **OTF_Writer_writeLeave** (**OTF_Writer** ∗writer, uint64_t time, uint32_t function, uint32_t process, uint32_t source)
- int **OTF_Writer_writeRecvMsg** (**OTF_Writer** ∗writer, uint64_t time, uint32_t receiver, uint32_t sender, uint32_t procGroup, uint32_t tag, uint32_t length, uint32_t source)
- int **OTF_Writer_writeSendMsg** (**OTF_Writer** ∗writer, uint64_t time, uint32_t sender, uint32_t receiver, uint32_t procGroup, uint32_t tag, uint32_t length, uint32_t source)
- int **OTF_Writer_writeCounter** (**OTF_Writer** ∗writer, uint64_t time, uint32_t process, uint32_t counter, uint64_t value)
- int **OTF_Writer_writeCollectiveOperation** (**OTF_Writer** ∗writer, uint64_t time, uint32_-t process, uint32_t collective, uint32_t procGroup, uint32_t rootProc, uint32_t sent, uint32_t received, uint64_t duration, uint32_t source)
- int **OTF_Writer_writeEventComment** (**OTF_Writer** ∗writer, uint64_t time, uint32_t process, const char ∗comment)
- int **OTF_Writer_writeBeginProcess** (**OTF_Writer** ∗writer, uint64_t time, uint32_t process)
- int **OTF_Writer_writeEndProcess** (**OTF_Writer** ∗writer, uint64_t time, uint32_t process)
- int **OTF_Writer_writeSnapshotComment** (**OTF_Writer** ∗writer, uint64_t time, uint32_-t process, const char ∗comment)
- int **OTF_Writer_writeEnterSnapshot** (**OTF_Writer** ∗writer, uint64_t time, uint64_t originaltime, uint32_t function, uint32_t process, uint32_t source)
- int **OTF_Writer_writeSendSnapshot** (**OTF_Writer** ∗writer, uint64_t time, uint64_t originaltime, uint32_t sender, uint32_t receiver, uint32_t procGroup, uint32_t tag, uint32_t source)
- int **OTF_Writer_writeSummaryComment** (**OTF_Writer** ∗writer, uint64_t time, uint32_-t process, const char ∗comment)
- int **OTF_Writer_writeFunctionSummary** (**OTF_Writer** ∗writer, uint64_t time, uint32_-t function, uint32_t process, uint64_t count, uint64_t excltime, uint64_t incltime)
- int **OTF_Writer_writeFunctionGroupSummary** (**OTF_Writer** ∗writer, uint64_t time, uint32_t functiongroup, uint32_t process, uint64_t count, uint64_t excltime, uint64_t incltime)
- int **OTF_Writer_writeMessageSummary** (**OTF_Writer** ∗writer, uint64_t time, uint32_t process, uint32_t peer, uint32_t comm, uint32_t tag, uint64_t number_sent, uint64_t number_-recved, uint64_t bytes_sent, uint64_t bytes_recved)

A.1 Function Documentation

A.1.1 OTF_Writer∗ OTF_Writer_open (char ∗ *fileNamePrefix*, uint32_t *numberOfStreams*, OTF_FileManager ∗ *fileManager*)

Create a new OTF_Writer instance with a given number of automatic streams.

Setting the number of streams to 0 causes the OTF_Writer object to create a separate stream for each process. Important! Explicit calls to **OTF_Writer_assignProcess()**(p. 23) can lead to an overall number of streams which exceeds the initial number of streams in this call. OTF can reduce its fi le handle usage to a given number. Therefore, an initialized fi le manager instance is needed as parameter. See OTF_FileManager for further details.

Parameters:

fileNamePrefix File name prefix which is going to be used by all sub-files which belong to the trace.

numberOfStreams Initial number of independent data streams to be generated.

fileManager File handle manager.

Returns:

Initialized OTF_Writer instance or 0 if a failure occurred.

A.1.2 int OTF_Writer_close (OTF_Writer ∗ *writer*)

Close an OTF_Writer instance and all its related files.

Parameters:

writer Pointer to an initialized OTF_Writer object. See also **OTF_Writer_open()**(p. 20).

Returns:

1 if instance was closed successfully and 0 otherwise.

A.1.3 int OTF_Writer_closeAllStreams (OTF_Writer ∗ *writer*)

Close all streams that are open in this writer instance.

Parameters:

writer Pointer to an initialized OTF_Writer object. See also **OTF_Writer_open()**(p. 20).

Returns:

1 on success, 0 if an error occurs.

A.1.4 int OTF_Writer_setCompression (OTF_Writer ∗ *writer*, OTF_FileCompression *compression*)

Set the standard compression method for all buffers managed by this writer

Parameters:

writer Pointer to an initialized OTF_Writer object. See also **OTF_Writer_open()**(p. 20).

compression compression method to apply to all following streams OTF_-FILECOMPRESSION_{UNCOMPRESSED, COMPRESSED}

Returns:

1 on success, 0 if an error occurs.

A.1.5 OTF_FileCompression OTF_Writer_getCompression (OTF_Writer ∗ *writer*)

Return the standard compression method for all buffers managed by this writer

Parameters:

writer Pointer to an initialized OTF_Writer object. See also **OTF_Writer_open()**(p. 20).

Returns:

Standard compression method for all buffers managed by this writer.

A.1.6 void OTF_Writer_setBufferSizes (OTF_Writer ∗ *writer*, uint32_t *size*)

Set the default buffer size for all buffers managed by this Writer. This is only effective for future buffers and will not change already allocated buffers. Those can be changed with the buffers directly.

Parameters:

writer Pointer to an initialized OTF_Writer object. See also **OTF_Writer_open()**(p. 20).

size Intended buffer size.

A.1.7 uint32_t OTF_Writer_getBufferSizes (OTF_Writer ∗ *writer*)

Get the default buffer size for all buffers managed by this Writer.

Parameters:

writer Pointer to an initialized OTF_Writer object. See also **OTF_Writer_open()**(p. 20).

Returns:

Default buffer size for all buffers managed by this Writer.

A.1.8 void OTF_Writer_setFormat (OTF_Writer ∗ *writer*, uint32_t *format*)

Set the default ouput format. The format is applied to all streams opened by the writer.

Parameters:

writer Pointer to an initialized OTF_Writer object. See also **OTF_Writer_open()**(p. 20).

format Intended output format (OTF_WSTREAM_FORMAT_{LONG,SHORT}).

A.1.9 uint32_t OTF_Writer_getFormat (OTF_Writer ∗ *writer*)

Get the default output format of all streams managed by this writer.

Parameters:

writer Pointer to an initialized OTF_Writer object. See also **OTF_Writer_open()**(p. 20).

Returns:

Default output format.

A.1.10 uint32_t OTF_Writer_assignProcess (OTF_Writer ∗ *writer*, uint32_t *process*, uint32_t *stream*)

Explicitly assign a given process to a specific stream.

Mind that 0 is not a valid stream or process identifier but a reserved value. By default, processes are automatically assigned to streams. Therefore, this call is optional.

Parameters:

writer Pointer to an initialized OTF_Writer object. See also **OTF_Writer_open()**(p. 20).

process Process identifier. See also **OTF_Writer_writeDefProcess()**(p. 24).

stream Target stream identifier with 0 < stream <= number of streams as defined in **OTF_-Writer_open()**(p. 20).

Returns:

1 on success, 0 if an error occurs.

A.1.11 OTF_MasterControl∗ OTF_Writer_getMasterControl (OTF_Writer ∗ *writer*)

Get a pointer to the master control object of the given writer instance.

Parameters:

writer Pointer to an initialized OTF_Writer object. See also **OTF_Writer_open()**(p. 20).

Returns:

Pointer to a master control object. See OTF_MasterControl.

A.1.12 void OTF_Writer_setMasterControl (OTF_Writer ∗ *writer*, OTF_MasterControl ∗ *mc*)

Set an alternative master control object. Use this only right after initialization but never after having written some records already!

Parameters:

writer Pointer to an initialized OTF_Writer object. See also **OTF_Writer_open()**(p. 20).

mc new master control object

A.1.13 int OTF_Writer_writeDefinitionComment (OTF_Writer ∗ *writer*, uint32_t *stream*, const char ∗ *comment*)

Write a comment record.

Parameters:

writer Initialized OTF_Writer instance.

stream Target stream identifier with 0 < stream <= number of streams as defined in **OTF_-Writer_open()**(p. 20).

comment Arbitrary comment string.

Returns:
1 on success, 0 if an error occurs.

A.1.14 int OTF_Writer_writeDefTimerResolution (OTF_Writer ∗ *writer*, uint32_t *stream*, uint64_t *ticksPerSecond*)

Write the timer resolution definition record. All timed event records will be interpreted according to this definition. By default, a timer resultion of 1 us i.e. 1,000,000 clock ticks is assumed.

Parameters:

writer Pointer to an initialized OTF_Writer object. See also **OTF_Writer_open()**(p. 20).

stream Target stream identifier with 0 < stream <= number of streams as defined in **OTF_-Writer_open()**(p. 20).

ticksPerSecond Clock ticks per second of the timer.

Returns:
1 on success, 0 if an error occurs.

A.1.15 int OTF_Writer_writeDefProcess (OTF_Writer ∗ *writer*, uint32_t *stream*, uint32_t *process*, const char ∗ *name*, uint32_t *parent*)

Write a process definition record.

Parameters:

writer Pointer to an initialized OTF_Writer object. See also **OTF_Writer_open()**(p. 20).

stream Target stream identifier with 0 < stream <= number of streams as defined in **OTF_-Writer_open()**(p. 20).

process Arbitrary but unique process identifier > 0.

name Name of the process e.g. "Process X".

parent Previously declared parent process identifier or 0 if process has no parent.

Returns:
1 on success, 0 if an error occurs.

A.1.16 int OTF_Writer_writeDefProcessGroup (OTF_Writer $*$ *writer*, uint32_t *stream*, uint32_t *procGroup*, const char $*$ *name*, uint32_t *numberOfProcs*, const uint32_t $*$ *procs*)

Write a process group deﬁ nition record.

OTF supports groups of processes. Their main objective is to classify processes depending on arbitrary characteristics. Processes can reside in multiple groups. This record type is optional.

Parameters:

writer Pointer to an initialized OTF_Writer object. See also **OTF_Writer_open()**(p. 20).

stream Target stream identiﬁ er with 0 $<$ stream $<=$ number of streams as deﬁ ned in **OTF_-Writer_open()**(p. 20).

procGroup Arbitrary but unique process group identiﬁ er $>$ 0.

name Name of the process group e.g. "Well Balanced".

numberOfProcs The number of processes in the process group.

procs Vector of process identiﬁ ers or previously deﬁ ned process group identiﬁ ers as deﬁ ned with **OTF_Writer_writeDefProcess()**(p. 24) resp. OTF_Writer_writeDefProcessGroup.

Returns:

1 on success, 0 if an error occurs.

A.1.17 int OTF_Writer_writeDefFunction (OTF_Writer $*$ *writer*, uint32_t *stream*, uint32_t *func*, const char $*$ *name*, uint32_t *funcGroup*, uint32_t *source*)

Write a function deﬁ nition record.

Deﬁ nes a function of the given name. Functions can optionally belong to a certain function group to be deﬁ ned with the **OTF_Writer_writeDefFunctionGroup()**(p. 26) call. A source code reference can be added to the deﬁ nition aswell.

Parameters:

writer Pointer to an initialized OTF_Writer object. See also **OTF_Writer_open()**(p. 20).

stream Target stream identiﬁ er with 0 $<$ stream $<=$ number of streams as deﬁ ned in **OTF_-Writer_open()**(p. 20).

func Arbitrary but unique function identiﬁ er $>$ 0.

name Name of the function e.g. "DoSomething".

funcGroup A function group identiﬁ er preliminary deﬁ ned with **OTF_Writer_writeDef-FunctionGroup()**(p. 26) or 0 for no function group assignment.

source Reference to the function's source code location preliminary deﬁ ned with **OTF_Writer_-writeDefScl()**(p. 27) or 0 for no source code location assignment.

Returns:

1 on success, 0 if an error occurs.

A.1.18 int OTF_Writer_writeDefFunctionGroup (OTF_Writer $*$ *writer*, uint32_t *stream*, uint32_t *funcGroup*, const char $*$ *name*)

Write a function group definition record.

Parameters:

writer Pointer to an initialized OTF_Writer object. See also **OTF_Writer_open()**(p. 20).

stream Target stream identifier with $0 <$ stream $<=$ number of streams as defined in **OTF_-Writer_open()**(p. 20).

funcGroup An arbitrary but unique function group identifier > 0.

name Name of the function group e.g. "Computation".

Returns:

1 on success, 0 if an error occurs.

A.1.19 int OTF_Writer_writeDefCollectiveOperation (OTF_Writer $*$ *writer*, uint32_t *stream*, uint32_t *collOp*, const char $*$ *name*, uint32_t *type*)

Write a collective operation definition record.

Parameters:

writer Initialized OTF_Writer instance.

stream Target stream identifier with $0 <$ stream $<=$ number of streams as defined in **OTF_-Writer_open()**(p. 20).

collOp An arbitrary but unique collective op. identifier > 0.

name Name of the collective operation e.g. "MPI_Bcast".

type One of the five supported collective classes: OTF_COLLECTIVE_TYPE_UNKNOWN (default), OTF_COLLECTIVE_TYPE_BARRIER, OTF_COLLECTIVE_TYPE_-ONE2ALL, OTF_COLLECTIVE_TYPE_ALL2ONE, OTF_COLLECTIVE_TYPE_-ALL2ALL.

Returns:

1 on success, 0 if an error occurs.

A.1.20 int OTF_Writer_writeDefCounter (OTF_Writer $*$ *writer*, uint32_t *stream*, uint32_t *counter*, const char $*$ *name*, uint32_t *properties*, uint32_t *counterGroup*, const char $*$ *unit*)

Write a counter definition record.

Parameters:

writer Initialized OTF_Writer instance.

stream Target stream identifier with $0 <$ stream $<=$ number of streams as defined in **OTF_-Writer_open()**(p. 20).

counter An arbitrary but unique counter identifi er.

name Name of the counter e.g. "Cache Misses".

properties A combination of a type and scope counter property. OTF_COUNTER_TYPE_ACC (default) represents a counter with monotonously increasing values e.g. a FLOP counter. OTF_COUNTER_TYPE_ABS on the other hand defi nes a counter with alternating absolute values e.g. the memory usage of a process. The following counter measurement scopes are supported: OTF_COUNTER_SCOPE_START (default) always refers to the start of the process, OTF_COUNTER_SCOPE_POINT refers to exactly this moment in time, OTF_-COUNTER_SCOPE_LAST relates to the previous measurement, and OTF_COUNTER_-SCOPE_NEXT to the next measurement. Examples: OTF_COUNTER_TYPE_ACC + OTF_COUNTER_SCOPE_START should be used for most standard hardware (PAPI) counters. OTF_COUNTER_TYPE_ABS + OTF_COUNTER_SCOPE_POINT could be used to record information 'spikes'. OTF_COUNTER_TYPE_ABS + OTF_COUNTER_-SCOPE_NEXT works for memory allocation recording.

counterGroup A previously defi ned counter group identifi er or 0 for no group.

unit Unit of the counter e.g. "#" for "number of..." or 0 for no unit.

Returns:

1 on success, 0 if an error occurs.

A.1.21 int OTF_Writer_writeDefCounterGroup (OTF_Writer ∗ *writer*, uint32_t *stream*, uint32_t *counterGroup*, const char ∗ *name*)

Write a counter group defi nition record.

Parameters:

writer Initialized OTF_Writer instance.

stream Target stream identifi er with 0 < stream <= number of streams as defi ned in **OTF_-Writer_open()**(p. 20).

counterGroup An arbitrary but unique counter group identifi er.

name Counter group name.

Returns:

1 on success, 0 if an error occurs.

A.1.22 int OTF_Writer_writeDefScl (OTF_Writer ∗ *writer*, uint32_t *stream*, uint32_t *source*, uint32_t *sourceFile*, uint32_t *line*)

Write a source code location (SCL) record.

Parameters:

writer Initialized OTF_Writer instance.

stream Target stream identifi er with 0 < stream <= number of streams as defi ned in **OTF_-Writer_open()**(p. 20).

source Arbitrary but unique source code location identifier > 0.

sourceFile Previously defined source file identifier. See **OTF_Writer_writeDefSclFile()**(p. 28).

line Line number.

Returns:

1 on success, 0 if an error occurs.

A.1.23 int OTF_Writer_writeDefSclFile (OTF_Writer $*$ *writer*, uint32_t *stream*, uint32_t *sourceFile*, const char $*$ *name*)

Write a source code location (SCL) file record.

Parameters:

writer Initialized OTF_Writer instance.

stream Target stream identifier with $0 <$ stream $<=$ number of streams as defined in **OTF_-Writer_open()**(p. 20).

sourceFile Arbitrary but unique source code location identifier != 0.

name File name.

Returns:

1 on success, 0 if an error occurs.

A.1.24 int OTF_Writer_writeOtfVersion (OTF_Writer $*$ *writer*, uint32_t *stream*)

Write a version record. There are no value parameters because the OTF version is determined by the currently used library itself.

Parameters:

writer Initialized OTF_Writer instance.

stream Target stream identifier with $0 <$ stream $<=$ number of streams as defined in **OTF_-Writer_open()**(p. 20).

Returns:

1 on success, 0 if an error occurs.

A.1.25 int OTF_Writer_writeDefCreator (OTF_Writer $*$ *writer*, uint32_t *stream*, const char $*$ *creator*)

Write a creator record.

Parameters:

writer Initialized OTF_Writer instance.

stream Target stream identifier with $0 <$ stream $<=$ number of streams as defined in **OTF_-Writer_open()**(p. 20).

creator String which identifies the creator of the file e.g. "TAU Version x.y.z".

Returns:

1 on success, 0 if an error occurs.

A.1.26 int OTF_Writer_writeEnter (OTF_Writer ∗ *writer*, uint64_t *time*, uint32_t *function*, uint32_t *process*, uint32_t *source*)

Write a function entry record.

Parameters:

writer Initialized OTF_Writer instance.

time The time when the function entry took place.

function Function to be entered as defi ned with OTF_Writer_defFunction.

process Process where action took place.

source Optional reference to source code.

Returns:

1 on success, 0 if an error occurs.

A.1.27 int OTF_Writer_writeLeave (OTF_Writer ∗ *writer*, uint64_t *time*, uint32_t *function*, uint32_t *process*, uint32_t *source*)

Write a function leave record.

Parameters:

writer Initialized OTF_Writer instance.

time The time when the function leave took place.

function Function which was left or 0 if stack integrety checking is not needed.

process Process where action took place.

source Explicit source code location or 0.

Returns:

1 on success, 0 if an error occurs.

A.1.28 int OTF_Writer_writeRecvMsg (OTF_Writer ∗ *writer*, uint64_t *time*, uint32_t *receiver*, uint32_t *sender*, uint32_t *procGroup*, uint32_t *tag*, uint32_t *length*, uint32_t *source*)

Write a message retrieval record.

Parameters:

writer Initialized OTF_Writer instance.

time The time when the message was received.

receiver Identifi er of receiving process.

sender Identifi er of sending process.

procGroup Optional process-group sender and receiver belong to, ’0’ for no group.

tag Optional message type information.

length Optional message length information.

source Optional reference to source code.

Returns:

1 on success, 0 if an error occurs.

A.1.29 int OTF_Writer_writeSendMsg (OTF_Writer ∗ *writer*, uint64_t *time*, uint32_t *sender*, uint32_t *receiver*, uint32_t *procGroup*, uint32_t *tag*, uint32_t *length*, uint32_t *source*)

Write a message send record.

Parameters:

writer Initialized OTF_Writer instance.

time The time when the message was send.

sender Sender of the message.

receiver Receiver of the message.

procGroup Optional process-group sender and receiver belong to, ’0’ for no group.

tag Optional message type information.

length Optional message length information.

source Optional reference to source code.

Returns:

1 on success, 0 if an error occurs.

A.1.30 int OTF_Writer_writeCounter (OTF_Writer ∗ *writer*, uint64_t *time*, uint32_t *process*, uint32_t *counter*, uint64_t *value*)

Write a counter measurement record.

Parameters:

writer Initialized OTF_Writer instance.

time Time when counter was measured.

process Process where counter measurment took place.

counter Counter which was measured.

value Counter value.

Returns:

1 on success, 0 if an error occurs.

A.1.31 int OTF_Writer_writeCollectiveOperation (OTF_Writer ∗ *writer*, uint64_t *time*, uint32_t *process*, uint32_t *collective*, uint32_t *procGroup*, uint32_t *rootProc*, uint32_t *sent*, uint32_t *received*, uint64_t *duration*, uint32_t *source*)

Write a collective operation member record.

Parameters:

writer Initialized OTF_Writer instance.

time Time when collective operation was entered by member.

process Process identifi er i.e. collective member.

collective Collective identifi er to be defi ned with **OTF_Writer_writeDefCollective-Operation()**(p. 26).

procGroup Group of processes participating in this collective.

rootProc Root process if != 0.

sent Data volume sent by member or 0.

received Data volumd received by member or 0.

duration Time spent in collective operation.

source Explicit source code location or 0.

Returns:

1 on success, 0 if an error occurs.

A.1.32 int OTF_Writer_writeEventComment (OTF_Writer ∗ *writer*, uint64_t *time*, uint32_t *process*, const char ∗ *comment*)

Write a comment record.

Parameters:

writer Initialized OTF_Writer instance.

time Comments need a timestamp for a proper positioning in the trace.

process Comments also need a process identifi er for a proper positioning in the trace.

comment Arbitrary comment string.

Returns:

1 on success, 0 if an error occurs.

A.1.33 int OTF_Writer_writeBeginProcess (OTF_Writer ∗ *writer*, uint64_t *time*, uint32_t *process*)

Write a begin process record

Parameters:

writer Initialized OTF_Writer instance.

time Time when process was referenced for the first time.

process Process identifier > 0.

Returns:

1 on success, 0 if an error occurs.

A.1.34 int OTF_Writer_writeEndProcess (OTF_Writer $*$ *writer*, uint64_t *time*, uint32_t *process*)

Write a end process record

Parameters:

writer Initialized OTF_Writer instance.

time Time when process was referenced for the last time.

process Process identifier > 0.

Returns:

1 on success, 0 if an error occurs.

A.1.35 int OTF_Writer_writeSnapshotComment (OTF_Writer $*$ *writer*, uint64_t *time*, uint32_t *process*, const char $*$ *comment*)

Write a snapshot comment record.

Parameters:

writer Initialized OTF_Writer instance.

time Comments need a timestamp for a proper positioning in the trace.

process Comments also need a process identifier for a proper positioning in the trace.

comment Arbitrary comment string.

Returns:

1 on success, 0 if an error occurs.

A.1.36 int OTF_Writer_writeEnterSnapshot (OTF_Writer $*$ *writer*, uint64_t *time*, uint64_t *originaltime*, uint32_t *function*, uint32_t *process*, uint32_t *source*)

Write an enter snapshot which provides information about a past function call

Parameters:

writer Initialized OTF_Writer instance.

time Time when the snapshot was written(current time).

originaltime Time when the according enter record was entered. This call is still on the stack.(It has not been left yet)

function Function that the has been entered OTF_Writer_defFunction.

process Process where action took place.

source Optional reference to source code.

Returns:

1 on success, 0 if an error occurs.

A.1.37 int OTF_Writer_writeSendSnapshot (OTF_Writer $*$ *writer*, uint64_t *time*, uint64_t *originaltime*, uint32_t *sender*, uint32_t *receiver*, uint32_t *procGroup*, uint32_t *tag*, uint32_t *source*)

Write a send snapshot which provides information about a past message send operation that is still pending, i.e. not yet received

Parameters:

writer Initialized OTF_Writer instance.

time Time when the snapshot was written(current time).

originaltime Time when the message was sent

sender Sender of the message.

receiver Receiver of the message.

procGroup Optional process-group sender and receiver belong to, ’0’ for no group.

tag Optional message type information.

source Optional reference to source code.

Returns:

1 on success, 0 if an error occurs.

A.1.38 int OTF_Writer_writeSummaryComment (OTF_Writer $*$ *writer*, uint64_t *time*, uint32_t *process*, const char $*$ *comment*)

Write a summary comment record.

Parameters:

writer Initialized OTF_Writer instance.

time Comments need a timestamp for a proper positioning in the trace.

process Comments also need a process identifi er for a proper positioning in the trace.

comment Arbitrary comment string.

Returns:

1 on success, 0 if an error occurs.

A.1.39 int OTF_Writer_writeFunctionSummary (OTF_Writer ∗ *writer*, uint64_t *time*, uint32_t *function*, uint32_t *process*, uint64_t *count*, uint64_t *excltime*, uint64_t *incltime*)

Write a function summary record.

Parameters:

writer Initialized OTF_Writer instance.

time Time when summary was computed.

function Function as defi ned with OTF_Handler_DefFunction.

process Process of the given function.

count Number of invocations.

excltime Time spent exclusively in the given function.

incltime Time spent in the given function including all sub-routine calls.

Returns:

1 on success, 0 if an error occurs.

A.1.40 int OTF_Writer_writeFunctionGroupSummary (OTF_Writer ∗ *writer*, uint64_t *time*, uint32_t *functiongroup*, uint32_t *process*, uint64_t *count*, uint64_t *excltime*, uint64_t *incltime*)

Write a functiongroup summary record.

Parameters:

writer Initialized OTF_Writer instance.

time Time when summary was computed.

functiongroup Function group as defi ned with OTF_Handler_DefFunctionGroup.

process Process of the given function group.

count Number of invocations.

excltime Time spent exclusively in the given function group.

incltime Time spent in the given function group including all sub-routine calls.

Returns:

1 on success, 0 if an error occurs.

A.1.41 int OTF_Writer_writeMessageSummary (OTF_Writer ∗ *writer*, uint64_t *time*, uint32_t *process*, uint32_t *peer*, uint32_t *comm*, uint32_t *tag*, uint64_t *number_sent*, uint64_t *number_recved*, uint64_t *bytes_sent*, uint64_t *bytes_recved*)

Write a message summary record.

Parameters:

writer Initialized OTF_Writer instance.

time Time when summary was computed.

process Process where messages originated.

peer Process where the message is sent to

comm Communicator of message summary

type Message type/tag.

number_sent The number of messages sent.

number_recved The number of messages received.

bytes_sent The number of bytes sent via messages of the given type.

bytes_recved The number of bytes received through messages of the given type.

Returns:

1 on success, 0 if an error occurs.

B Global Trace Record Input Handler Interface

Functions

- int **OTF_Handler_DefinitionComment** (void ∗userData, uint32_t stream, const char ∗comment)
- int **OTF_Handler_DefTimerResolution** (void ∗userData, uint32_t stream, uint64_t ticksPer-Second)
- int **OTF_Handler_DefProcess** (void ∗userData, uint32_t stream, uint32_t process, const char ∗name, uint32_t parent)
- int **OTF_Handler_DefProcessGroup** (void ∗userData, uint32_t stream, uint32_t procGroup, const char ∗name, uint32_t numberOfProcs, const uint32_t ∗procs)
- int **OTF_Handler_DefFunction** (void ∗userData, uint32_t stream, uint32_t func, const char ∗name, uint32_t funcGroup, uint32_t source)
- int **OTF_Handler_DefFunctionGroup** (void ∗userData, uint32_t stream, uint32_t funcGroup, const char ∗name)
- int **OTF_Handler_DefCollectiveOperation** (void ∗userData, uint32_t stream, uint32_t coll-Op, const char ∗name, uint32_t type)
- int **OTF_Handler_DefCounter** (void ∗userData, uint32_t stream, uint32_t counter, const char ∗name, uint32_t properties, uint32_t counterGroup, const char ∗unit)
- int **OTF_Handler_DefCounterGroup** (void ∗userData, uint32_t stream, uint32_t counter-Group, const char ∗name)
- int **OTF_Handler_DefScl** (void ∗userData, uint32_t stream, uint32_t source, uint32_t source-File, uint32_t line)
- int **OTF_Handler_DefSclFile** (void ∗userData, uint32_t stream, uint32_t sourceFile, const char ∗name)
- int **OTF_Handler_DefCreator** (void ∗userData, uint32_t stream, const char ∗creator)
- int **OTF_Handler_Enter** (void ∗userData, uint64_t time, uint32_t function, uint32_t process, uint32_t source)
- int **OTF_Handler_Leave** (void ∗userData, uint64_t time, uint32_t function, uint32_t process, uint32_t source)
- int **OTF_Handler_SendMsg** (void ∗userData, uint64_t time, uint32_t sender, uint32_t re-ceiver, uint32_t group, uint32_t type, uint32_t length, uint32_t source)
- int **OTF_Handler_RecvMsg** (void ∗userData, uint64_t time, uint32_t recvProc, uint32_t send-Proc, uint32_t group, uint32_t type, uint32_t length, uint32_t source)
- int **OTF_Handler_Counter** (void ∗userData, uint64_t time, uint32_t process, uint32_t counter, uint64_t value)
- int **OTF_Handler_CollectiveOperation** (void ∗userData, uint64_t time, uint32_t process, uint32_t collective, uint32_t procGroup, uint32_t rootProc, uint32_t sent, uint32_t received, uint64_t duration, uint32_t source)
- int **OTF_Handler_EventComment** (void ∗userData, uint64_t time, uint32_t process, const char ∗comment)
- int **OTF_Handler_BeginProcess** (void ∗userData, uint64_t time, uint32_t process)
- int **OTF_Handler_EndProcess** (void ∗userData, uint64_t time, uint32_t process)

- int **OTF_Handler_SnapshotComment** (void ∗userData, uint64_t time, uint32_t process, const char ∗comment)
- int **OTF_Handler_EnterSnapshot** (void ∗userData, uint64_t time, uint64_t originaltime, uint32_t function, uint32_t process, uint32_t source)
- int **OTF_Handler_SendSnapshot** (void ∗userData, uint64_t time, uint64_t originaltime, uint32_t sender, uint32_t receiver, uint32_t procGroup, uint32_t tag, uint32_t source)
- int **OTF_Handler_SummaryComment** (void ∗userData, uint64_t time, uint32_t process, const char ∗comment)
- int **OTF_Handler_FunctionSummary** (void ∗userData, uint64_t time, uint32_t function, uint32_t process, uint64_t invocations, uint64_t exclTime, uint64_t inclTime)
- int **OTF_Handler_FunctionGroupSummary** (void ∗userData, uint64_t time, uint32_t funcGroup, uint32_t process, uint64_t invocations, uint64_t exclTime, uint64_t inclTime)
- int **OTF_Handler_MessageSummary** (void ∗userData, uint64_t time, uint32_t process, uint32_t peer, uint32_t comm, uint32_t type, uint64_t sentNumber, uint64_t receivedNumber, uint64_t sentBytes, uint64_t receivedBytes)
- int **OTF_Handler_UnknownRecord** (void ∗userData, uint64_t time, uint32_t process, const char ∗record)

B.1 Detailed Description

In the following, the handler interfaces for all record types are specifi ed. The signature of callback handler functions is equal to the signature of corresponding record write functions except for the fi rst argument. The fi rst argument common to all callback handler functions is *userData* – a generic pointer to custom user data. The second common argument to all callback hander functions is *stream* which identifi es the stream where the defi nition occurred. A stream parameter = 0 indicates a global defi nition which is the default.

B.2 Function Documentation

B.2.1 int OTF_Handler_DefinitionComment (void ∗ *userData*, uint32_t *stream*, const char ∗ *comment*)

Provides a comment record.

Parameters:

userData Pointer to user data which can be set with **OTF_HandlerArray_setFirstHandlerArg()**(p. 16).

stream Identifi es the stream to which this defi nition belongs to. stream = 0 represents a global defi nition.

comment Arbitrary comment string.

Returns:

’1’ for aborting the reading process immediately ’0’ for continue reading

B.2.2 int OTF_Handler_DefTimerResolution (void * *userData*, uint32_t *stream*, uint64_t *ticksPerSecond*)

Provides the timer resolution. All timed event records need to be interpreted according to this definition. By default, a timer resolution of 1 us i.e. 1,000,000 clock ticks is assumed.

Parameters:

userData Pointer to user data which can be set with **OTF_HandlerArray_setFirstHandlerArg()**(p. 16).

stream Identifies the stream to which this definition belongs to. stream = 0 represents a global definition.

ticksPerSecond Clock ticks per second of the timer.

Returns:

'1' for aborting the reading process immediately '0' for continue reading

B.2.3 int OTF_Handler_DefProcess (void * *userData*, uint32_t *stream*, uint32_t *process*, const char * *name*, uint32_t *parent*)

Provides a process definition.

Parameters:

userData Pointer to user data which can be set with **OTF_HandlerArray_setFirstHandlerArg()**(p. 16).

stream Identifies the stream to which this definition belongs to. stream = 0 represents a global definition.

process Arbitrary but unique process identifier > 0.

name Name of the process e.g. "Process X".

parent Previously declared parent process identifier or 0 if process has no parent.

Returns:

'1' for aborting the reading process immediately '0' for continue reading

B.2.4 int OTF_Handler_DefProcessGroup (void * *userData*, uint32_t *stream*, uint32_t *procGroup*, const char * *name*, uint32_t *numberOfProcs*, const uint32_t * *procs*)

Provides a process group definition.

OTF supports groups of processes. Their main objective is to classify processes depending on arbitrary characteristics. Processes can reside in multiple groups. This record type is optional.

Parameters:

userData Pointer to user data which can be set with **OTF_HandlerArray_setFirstHandlerArg()**(p. 16).

stream Identifies the stream to which this definition belongs to. stream = 0 represents a global definition.

procGroup Arbitrary but unique process group identifier > 0.

name Name of the process group e.g. "Well Balanced".

numberOfProcs The number of processes in the process group.

procs Vector of process identifiers as provided by **OTF_Handler_DefProcess()**(p. 38).

Returns:

'1' for aborting the reading process immediately '0' for continue reading

B.2.5 int OTF_Handler_DefFunction (void $*$ *userData*, uint32_t *stream*, uint32_t *func*, const char $*$ *name*, uint32_t *funcGroup*, uint32_t *source*)

Provides a function definition.

Defines a function of the given name. Functions can optionally belong to a certain function group provided by the **OTF_Handler_DefFunctionGroup()**(p. 39) handler. A source code reference can be provided aswell.

Parameters:

userData Pointer to user data which can be set with **OTF_HandlerArray_setFirstHandler-Arg()**(p. 16).

stream Identifies the stream to which this definition belongs to. stream = 0 represents a global definition.

func Arbitrary but unique function identifier > 0.

name Name of the function e.g. "DoSomething".

funcGroup A function group identifier preliminary provided by **OTF_Handler_DefFunction-Group()**(p. 39) or 0 for no function group assignment.

source Reference to the function's source code location preliminary provided by **OTF_-Handler_DefScl()**(p. 41) or 0 for no source code location assignment.

Returns:

'1' for aborting the reading process immediately '0' for continue reading

B.2.6 int OTF_Handler_DefFunctionGroup (void $*$ *userData*, uint32_t *stream*, uint32_t *funcGroup*, const char $*$ *name*)

Provides a function group definition.

Parameters:

userData Pointer to user data which can be set with **OTF_HandlerArray_setFirstHandler-Arg()**(p. 16).

stream Identifies the stream to which this definition belongs to. stream = 0 represents a global definition.

funcGroup An arbitrary but unique function group identifier > 0.

name Name of the function group e.g. "Computation".

Returns:

'1' for aborting the reading process immediately '0' for continue reading

B.2.7 int OTF_Handler_DefCollectiveOperation (void $*$ *userData*, uint32_t *stream*, uint32_t *collOp*, const char $*$ *name*, uint32_t *type*)

Provides a collective operation definition.

Parameters:

userData Pointer to user data which can be set with **OTF_HandlerArray_setFirstHandler-Arg()**(p. 16).

stream Identifies the stream to which this definition belongs to. stream = 0 represents a global definition.

collOp An arbitrary but unique collective op. identifier > 0.

name Name of the collective operation e.g. "MPI_Bcast".

type One of the five supported collective classes: OTF_COLLECTIVE_TYPE_UNKNOWN (default), OTF_COLLECTIVE_TYPE_BARRIER, OTF_COLLECTIVE_TYPE_-ONE2ALL, OTF_COLLECTIVE_TYPE_ALL2ONE, OTF_COLLECTIVE_TYPE_-ALL2ALL.

Returns:

'1' for aborting the reading process immediately '0' for continue reading

B.2.8 int OTF_Handler_DefCounter (void $*$ *userData*, uint32_t *stream*, uint32_t *counter*, const char $*$ *name*, uint32_t *properties*, uint32_t *counterGroup*, const char $*$ *unit*)

Provides a counter definition.

Parameters:

userData Pointer to user data which can be set with **OTF_HandlerArray_setFirstHandler-Arg()**(p. 16).

stream Identifies the stream to which this definition belongs to. stream = 0 represents a global definition.

counter An arbitrary but unique counter identifier.

name Name of the counter e.g. "Cache Misses".

properties A combination of a type and scope counter property. OTF_COUNTER_TYPE_ACC (default) represents a counter with monotonously increasing values e.g. a FLOP counter. OTF_COUNTER_TYPE_ABS on the other hand defines a counter with alternating absolute values e.g. the memory usage of a process. The following counter measurement scopes are supported: OTF_COUNTER_SCOPE_START (default) always refers to the start of the process, OTF_COUNTER_SCOPE_POINT refers to exactly this moment in time, OTF_-COUNTER_SCOPE_LAST relates to the previous measurement, and OTF_COUNTER_-SCOPE_NEXT to the next measurement. Examples: OTF_COUNTER_TYPE_ACC + OTF_COUNTER_SCOPE_START should be used for most standard hardware (PAPI) counters. OTF_COUNTER_TYPE_ABS + OTF_COUNTER_SCOPE_POINT could be used to record information 'spikes'. OTF_COUNTER_TYPE_ABS + OTF_COUNTER_-SCOPE_NEXT works for memory allocation recording.

counterGroup A previously defined counter group identifier or 0 for no group.

unit Unit of the counter e.g. "#" for "number of..." or 0 for no unit.

Returns:

'1' for aborting the reading process immediately '0' for continue reading

B.2.9 int OTF_Handler_DefCounterGroup (void ∗ *userData*, uint32_t *stream*, uint32_t *counterGroup*, const char ∗ *name*)

Provides a counter group definition.

Parameters:

userData Pointer to user data which can be set with **OTF_HandlerArray_setFirstHandler-Arg()**(p. 16).

stream Identifies the stream to which this definition belongs to. stream = 0 represents a global definition.

counterGroup An arbitrary but unique counter group identifier > 0.

name Counter group name.

Returns:

'1' for aborting the reading process immediately '0' for continue reading

B.2.10 int OTF_Handler_DefScl (void ∗ *userData*, uint32_t *stream*, uint32_t *source*, uint32_t *sourceFile*, uint32_t *line*)

Provides a source code location (SCL).

Parameters:

userData Pointer to user data which can be set with **OTF_HandlerArray_setFirstHandler-Arg()**(p. 16).

stream Identifies the stream to which this definition belongs to. stream = 0 represents a global definition.

source Arbitrary but unique source code location identifier > 0.

sourceFile Previously defined source file identifier. See OTW_Handler_DefSclFile().

line Line number.

Returns:

'1' for aborting the reading process immediately '0' for continue reading

B.2.11 int OTF_Handler_DefSclFile (void $*$ *userData*, uint32_t *stream*, uint32_t *sourceFile*, const char $*$ *name*)

Provides a source code location (SCL) fi le.

Parameters:

userData Pointer to user data which can be set with **OTF_HandlerArray_setFirstHandlerArg()**(p. 16).

stream Identifi es the stream to which this defi nition belongs to. stream = 0 represents a global defi nition.

sourceFile Arbitrary but unique source code location identifi er != 0.

name File name.

Returns:

’1’ for aborting the reading process immediately ’0’ for continue reading

B.2.12 int OTF_Handler_DefCreator (void $*$ *userData*, uint32_t *stream*, const char $*$ *creator*)

Provides fi le creator information.

Parameters:

userData Pointer to user data which can be set with **OTF_HandlerArray_setFirstHandlerArg()**(p. 16).

stream Identifi es the stream to which this defi nition belongs to. stream = 0 represents a global defi nition.

creator String which identifi es the creator of the fi le e.g. "TAU Version x.y.z".

Returns:

’1’ for aborting the reading process immediately ’0’ for continue reading

B.2.13 int OTF_Handler_Enter (void $*$ *userData*, uint64_t *time*, uint32_t *function*, uint32_t *process*, uint32_t *source*)

Provides a function entry event.

Parameters:

userData Pointer to user data which can be set with **OTF_HandlerArray_setFirstHandlerArg()**(p. 16).

time The time when the function entry took place.

function Function which has been entered as defi ned with OTF_Writer_defFunction.

process Process where action took place.

source Explicit source code location identifi er $>$ 0 or 0 if no source information available.

Returns:

’1’ for aborting the reading process immediately ’0’ for continue reading

B.2.14 int OTF_Handler_Leave (void $*$ *userData*, uint64_t *time*, uint32_t *function*, uint32_t *process*, uint32_t *source*)

Provides a function leave event.

Parameters:

userData Pointer to user data which can be set with **OTF_HandlerArray_setFirstHandler-Arg()**(p. 16).

time The time when the function leave took place.

function Function which was left or 0 if stack integrety checking is not available.

process Process where action took place.

source Explicit source code location identifi er $>$ 0 or 0 if no source information available.

Returns:

’1’ for aborting the reading process immediately ’0’ for continue reading

B.2.15 int OTF_Handler_SendMsg (void $*$ *userData*, uint64_t *time*, uint32_t *sender*, uint32_t *receiver*, uint32_t *group*, uint32_t *type*, uint32_t *length*, uint32_t *source*)

Provides a message send event.

Parameters:

userData Pointer to user data which can be set with **OTF_HandlerArray_setFirstHandler-Arg()**(p. 16).

time The time when the message was send.

sender Sender of the message.

receiver Receiver of the message.

group Process-group to which sender and receiver belong to or 0 for no group assignment.

type Message type information $>$ 0 or 0 for no information.

length Optional message length information.

source Explicit source code location identifi er $>$ 0 or 0 if no source information available.

Returns:

’1’ for aborting the reading process immediately ’0’ for continue reading

B.2.16 int OTF_Handler_RecvMsg (void $*$ *userData*, uint64_t *time*, uint32_t *recvProc*, uint32_t *sendProc*, uint32_t *group*, uint32_t *type*, uint32_t *length*, uint32_t *source*)

Provides a message retrieval event.

Parameters:

userData Pointer to user data which can be set with **OTF_HandlerArray_setFirstHandler-Arg()**(p. 16).

time The time when the message was received.

recvProc Identifi er of receiving process.

sendProc Identifi er of sending process.

group Process-group to which sender and receiver belong to or 0 for no group assignment.

type Message type information > 0 or 0 for no information.

length Optional message length information.

source Explicit source code location identifi er > 0 or 0 if no source information available.

Returns:

’1’ for aborting the reading process immediately ’0’ for continue reading

B.2.17 int OTF_Handler_Counter (void $*$ *userData*, uint64_t *time*, uint32_t *process*, uint32_t *counter*, uint64_t *value*)

Provides a counter measurement.

Parameters:

userData Pointer to user data which can be set with **OTF_HandlerArray_setFirstHandler-Arg()**(p. 16).

time Time when counter was measured.

process Process where counter measurment took place.

counter Counter which was measured.

value Counter value.

Returns:

’1’ for aborting the reading process immediately ’0’ for continue reading

B.2.18 int OTF_Handler_CollectiveOperation (void $*$ *userData*, uint64_t *time*, uint32_t *process*, uint32_t *collective*, uint32_t *procGroup*, uint32_t *rootProc*, uint32_t *sent*, uint32_t *received*, uint64_t *duration*, uint32_t *source*)

Provides a collective operation member event.

Parameters:

userData Pointer to user data which can be set with **OTF_HandlerArray_setFirstHandler-Arg()**(p. 16).

time Time when collective operation was entered by member.

process Process identifi er i.e. collective member.

collective Collective identifi er as defi ned with OTF_Handler_eDefCollectiveOperation().

procGroup Group of processes participating in this collective.

rootProc Root process if != 0.

sent Data volume sent by member or 0.

received Data volume received by member or 0.

duration Time spent in collective operation.

source Explicit source code location or 0.

Returns:

'1' for aborting the reading process immediately '0' for continue reading

B.2.19 int OTF_Handler_EventComment (void $*$ *userData*, uint64_t *time*, uint32_t *process*, const char $*$ *comment*)

Provide a comment record.

Parameters:

userData Pointer to user data which can be set with **OTF_HandlerArray_setFirstHandler-Arg()**(p. 16).

time Comments need a timestamp for a proper positioning in the trace.

process Comments also need a process identifi er for a proper positioning in the trace.

comment Arbitrary comment string.

Returns:

'1' for aborting the reading process immediately '0' for continue reading

B.2.20 int OTF_Handler_BeginProcess (void $*$ *userData*, uint64_t *time*, uint32_t *process*)

Provides a process creation event.

Marks the explicit begin of a process. This event precedes the very fi rst event of the respective process and should carry the same time stamp. This is especially useful with on-line analysis. It tells whether there will be additional records for the given process or not. Without this record type, it could only be guessed that there might not follow more events after a process has reached the bottom of the call stack.

Parameters:

userData Pointer to user data which can be set with **OTF_HandlerArray_setFirstHandler-Arg()**(p. 16).

time Time when process was referenced for the fi rst time.

process Process identifi er > 0.

Returns:

'1' for aborting the reading process immediately '0' for continue reading

B.2.21 int OTF_Handler_EndProcess (void $*$ *userData*, uint64_t *time*, uint32_t *process*)

Provides a process destruction event.

Parameters:

userData Pointer to user data which can be set with **OTF_HandlerArray_setFirstHandler-Arg()**(p. 16).

time Time when process is referenced for the last time. Process identifi ers must not be recycled!

process Process identifi er > 0.

Returns:

'1' for aborting the reading process immediately '0' for continue reading

B.2.22 int OTF_Handler_SnapshotComment (void $*$ *userData*, uint64_t *time*, uint32_t *process*, const char $*$ *comment*)

Provides a snapshot comment.

Parameters:

userData Pointer to user data which can be set with **OTF_HandlerArray_setFirstHandler-Arg()**(p. 16).

time Comments need a timestamp for a proper positioning in the trace.

process Comments also need a process identifi er for a proper positioning in the trace.

comment Arbitrary comment string.

Returns:

'1' for aborting the reading process immediately '0' for continue reading

B.2.23 int OTF_Handler_EnterSnapshot (void $*$ *userData*, uint64_t *time*, uint64_t *originaltime*, uint32_t *function*, uint32_t *process*, uint32_t *source*)

provides information about a past function call at the time 'originaltime'. Parameters 'time', 'function', 'process' ,'source' and the return value have the same meaning as in **OTF_Handler_-Enter()**(p. 42).

B.2.24 int OTF_Handler_SendSnapshot (void $*$ *userData*, uint64_t *time*, uint64_t *originaltime*, uint32_t *sender*, uint32_t *receiver*, uint32_t *procGroup*, uint32_t *tag*, uint32_t *source*)

provides information about a past message send operation at the time 'originaltime'. Parameters 'time', 'sender', 'receiver', 'procGroup', 'tag', 'source' and the return value have the same meaning as in **OTF_Handler_SendMsg()**(p. 43).

B.2.25 int OTF_Handler_SummaryComment (void ∗ *userData*, uint64_t *time*, uint32_t *process*, const char ∗ *comment*)

Provides a summary comment.

Parameters:

userData Pointer to user data which can be set with **OTF_HandlerArray_setFirstHandlerArg()**(p. 16).

time Comments need a timestamp for a proper positioning in the trace.

process Comments also need a process identifi er for a proper positioning in the trace.

comment Arbitrary comment string.

Returns:

’1’ for aborting the reading process immediately ’0’ for continue reading

B.2.26 int OTF_Handler_FunctionSummary (void ∗ *userData*, uint64_t *time*, uint32_t *function*, uint32_t *process*, uint64_t *invocations*, uint64_t *exclTime*, uint64_t *inclTime*)

Provides summarized information for a given function.

Parameters:

userData Pointer to user data which can be set with **OTF_HandlerArray_setFirstHandlerArg()**(p. 16).

time Time when summary was computed.

function Function as defi ned with OTF_Handler_DefFunction.

process Process of the given function.

invocations Number of invocations.

exclTime Time spent exclusively in the given function.

inclTime Time spent in the given function including all sub-routine calls.

Returns:

’1’ for aborting the reading process immediately ’0’ for continue reading

B.2.27 int OTF_Handler_FunctionGroupSummary (void ∗ *userData*, uint64_t *time*, uint32_t *funcGroup*, uint32_t *process*, uint64_t *invocations*, uint64_t *exclTime*, uint64_t *inclTime*)

Provides summarized information for a given group of functiongroups.

Parameters:

userData Pointer to user data which can be set with **OTF_HandlerArray_setFirstHandlerArg()**(p. 16).

time Time when summary was computed.

funcGroup Function group as defined with OTF_Handler_DefFunctionGroup.

process Process of the given function group.

invocations Number of invocations.

exclTime Time spent exclusively in the given function group.

inclTime Time spent in the given function group including all sub-routine calls.

Returns:
’1’ for aborting the reading process immediately ’0’ for continue reading

B.2.28 int OTF_Handler_MessageSummary (void ∗ *userData*, uint64_t *time*, uint32_t *process*, uint32_t *peer*, uint32_t *comm*, uint32_t *type*, uint64_t *sentNumber*, uint64_t *receivedNumber*, uint64_t *sentBytes*, uint64_t *receivedBytes*)

Provides summarized information for a given message type.

Parameters:

userData Pointer to user data which can be set with **OTF_HandlerArray_setFirstHandlerArg()**(p. 16).

time Time when summary was computed.

process Process where messages originated.

peer Process where the message is sent to

comm Communicator of message summary

type Message type/tag.

sentNumber The number of messages sent.

receivedNumber The number of messages received.

sentBytes The number of bytes sent via messages of the given type.

receivedBytes The number of bytes received through messages of the given type.

Returns:
’1’ for aborting the reading process immediately ’0’ for continue reading

B.2.29 int OTF_Handler_UnknownRecord (void ∗ *userData*, uint64_t *time*, uint32_t *process*, const char ∗ *record*)

Can be used to handle records which cannot be read.

Parameters:

userData Pointer to user data which can be set with **OTF_HandlerArray_setFirstHandlerArg()**(p. 16).

time Time when summary was computed.

process If ’time’ equals (uin64_t) -1, the unknown record is a defi niton record and ’process’ represents the streamid of the record. If ’time’ has a valid value (not (uint64)-1) the unknown record is an event-, statistics- or snapshotrecord and ’process’ represents the processid of the record.

record string which contains the record.

Returns:

’1’ for aborting the reading process immediately ’0’ for continue reading

C Changelog

```
1.0.x
- initial version

1.1.1
- OTF_Reader now considers the return values of the handlers
- added OTF_VERBOSE, OTF_DEBUG macro for error treatment
- introduced ’UnknownRecord’ handler which allows to catch
  unknown record types

1.1.2
- inverted return value of call-back handlers:
    ’0’ is non-error, ’!= 0’ is regarded as an error, now!
    (this makes OTF conform with the VTF3 scheme.)

1.1.3
- fixed a minor bug in otfaux

1.1.4
- fixed a bug in OTF_Reader which might have caused the very first
time stamp of a trace to be not properly sorted
- introduced ’--snapshots’ and ’--statistics’ switches to do only
snapshots or statistics. for statistics a selective mode is allowed
which regards only some streams. By this means statistics can be created
in parallel by calling otfaux multiple times.

1.1.5
- have UnknownRecord report handle incomplete records or additional bytes at
the end of a file.

1.2.0
- introduce transparent zlib compression

1.2.1
- added progress functions using read bytes instead of timestamps

1.2.2
- important bugfix: definitionstream 0 was ignored since version 1.2.0

1.2.3
- bugfix: provided copy handlers returned wrong value

1.2.4
- bugfix: zlib compression bug, wrong sanity check fixed
```