ZHR–R–0304

VTF3 – A Fast Vampir Trace File Low-Level Management Library

Stephan Seidl

Dresden University of Technology
Center for High-Performance Computing
Postfach 01062 Dresden, Germany
`seidl@zhr.tu-dresden.de`
`http://www.tu-dresden.de/zhr`

November 17, 2003

Abstract. This document describes VTF3-1.37 for writing and reading traditional VAMPIR [1] trace files. VTF3 is part of newer VAMPIR versions and manages the lowest level of the prime source channel there. Developed in 2001, it can be used separately and freely. Except for some records, being part of a research project, all the C-interfaces are defined to be frozen. In addition to pure interface descriptions, the document provides semantic background and gives a lot of hints so that, for example, converter developers get the information they need to imagine what VAMPIR expects as input. Besides their visualization, VTF3-style all-in-one trace files are more and more often used to comfortably store, process and interchange general temporal states and functions, with or without a high degree of parallelism.

1 Introduction

VAMPIR 3.0 [2] supports two source types. These are the new STF (Structured Trace File) type, which has been designed and implemented by PALLAS GmbH, and the traditional VTF (VAMPIR Trace File) type. VTF3 is the software to handle the VTF branch. In 2001, VTF3 was developed from scratch to be a fast, portable, tight and clear layer to cover all the VTF management requirements. A lot of new records have been introduced, some of them to improve speed and clarity only. On the other hand, VTF3 accepts the whole trace file history as input. The files may be plain or compressed; compression/uncompression is done on the fly between memory buffers using VTF3-ZLIB-1.1.4. VTF3-ZLIB-1.1.4 has been derived from zlib-1.1.4 by Jean-loup Gailly, Mark Adler and others.

Traditional VAMPIR VTF3-style all-in-one trace files are record-based and are subdivided into two sections, being the declaration section on top and the following event record section. All the record entries of each section have a global scope. Especially with respect to the declaration records, it is no longer allowed to overwrite any prior defined one, even if the new record exactly reproduces an old declaration. There is no reason to have two or more declaration records carrying the same information. The records of the declaration section do not have a timestamp; the records of the event record section have one.

A VTF may be sorted or unsorted. Unsorted files cannot directly be processed by Vampir. An unsorted state should be flagged by an *Unmerged* record as the absolutely first one. There is a special VTF3 application, called Vptmerge, by Sven Bauer, which is able to merge sorted or unsorted VTF fragments to create a sorted, all-in-one trace file readable by Vampir. Vptmerge can also be used to change from one of the four VTF formats into another, although the legacy VTF ASCII format is not a possible output option. These four VTF formats are the *Standard Binary Format*, the *Standard ASCII Format*, the *Fast ASCII Format* and the *Legacy ASCII Format*. All of them are portable, i.e., they can be moved from one architecture to another without changes. The first three formats have a file header on top, the *Legacy ASCII Format* has none. For the Unix *file(1)* command, appropriate `/etc/magic` entries would look as described in table 1.

Table 1. `/etc/magic` *(magic(5))* entries for *file(1)* version 3.27

```
0 string U\252\245ZBPVI Vampir Standard-Binary trace
0 string C\040SAVTF\n   Vampir Standard-ASCII trace
0 string C\040SAVTF\r\n Vampir Standard-ASCII trace
0 string C\040FAVTF\n   Vampir Fast-ASCII trace
0 string C\040FAVTF\r\n Vampir Fast-ASCII trace
```

Besides using Vptmerge, VTF fragments, being each of the same format, can also be concatenated with the help of operating system utilities. File headers within VTFs are recognized and automatically skipped by the VTF3 scanners. However, under certain non-Unix operating systems, using the *Standard Binary Format*, one has to ensure that all these copy actions are performed in binary mode. The ASCII formats resist the usual changes, for example, the occurrence of `^M-^J`-sequences instead of single new-line characters.

The current version of Vptmerge, which is also based on VTF3-1.37, filters out *Unmerged* records of any kind, removes declaration section record duplicates, and sorts the records of this section group-wise according to table 2. Moreover, it replaces some outmoded records by more efficient ones for significantly better performance. Back to the trace file structure. An application, which wants to create trace files ready for Vampir, perhaps uses the Vptmerge group sort order, because this order is safe.

Except for the record type groups *Defcpugrp* and *Defpatternshape*, tagged with (•) in table 2, the record sort order is irrelevant within the same group. In contrast, the records of the groups with a (•) have to be sorted so that

Table 2. The declaration section record type groups in safe order

1. *Defversion*	11. *Defact*	21. *Defcommunicator*
2. *Defcreator*	12. *Defstate*	22. *Defmsgname*
3. *Defsyscpunums*	13. *Defact_obsol* (†)	23. *Defglobalop*
4. *Defsyscpunames* (†)	14. *Defstate_obsol* (†)	24. *Defredfunc_obsol* (†)
5. *Defthreadnums*	15. *Defcpuregclass* (†)	25. *Defiofile*
6. *Defcpuname*	16. *Defclstrregclass* (†)	26. *Defkparreg*
7. *Defclstr* (†)	17. *Defsampclass*	27. *Defclkperiod*
8. *Defcpugrp* (●)	18. *Defcpureg* (†)	28. *Deftimeoffset*
9. *Defsclfile*	19. *Defclstrreg* (†)	29. *Defpatternshape* (●)
10. *Defscl*	20. *Defsamp*	30. *Defpattern*

all the references are back references. Vptmerge takes this into account, too. Declaration record types tagged with (†) are outmoded.

The records of the event section, being the second and last one after the declaration section, carry a timestamp multiplier and have to be chronologically sorted for Vampir. Timestamp multipliers are non-negative integer numbers out of the interval $[0, 10^{30}]$. Though they are passed through the interfaces as C-language *double* types, their fractional part is filtered out by rounding to keep the binary and ASCII formats in strict conformance to each other.

2 Writing Vampir Trace Files

Vampir trace files are principally created by so-called trace libraries. Since tracing always changes the run-time characteristics of the researched subject, it is crucial to keep the overhead losses as small as possible. Good trace libraries do not need more than a few hundred clocks to pick up a timestamp together with some performance counters and to rawly deposit all these data in a trace buffer. If the trace buffer overflows, tracing may be switched off, or a flushing routine writes its contents onto the disk. The latter operation may be done rawly or translating the data into the final format. We touch on this topic here to clarify the position of VTF3 inside a tracing environment. Even though VTF3-1.37 is a fast software, it cannot be used to deposit program trace data in the buffer; it should be used to bring out the raw trace buffer contents onto the disk, properly formatted, to get the final Vampir input or appropriate fragments, respectively. Fragments can be post-processed by hand or with the help of Vptmerge.

Hence, the layer on top of VTF3 will collect information needed to write all the VTF records, one after another, and will release this information invoking the VTF3 interfaces. Furthermore, this layer should control the size of the resulting files. Without compression, handy trace files do not exceed 50 MB, but, of course, Vampir visualizes files of several hundred megabytes without any problem as long as one has enough main memory installed, e.g. 1 GB.

When VTF3-1.37 is used for creating trace files, the Vampir version has to match, i.e., the VTF3 release number of this Vampir cannot be smaller than 1.37. One should have a look at the `Handled_by:`-line of the VAMPIR `File.Tracefile_Info` display or should simply enter

```
strings ./vampir | grep vtf3
```

at a UNIX/LINUX command prompt.

```
#include <stddef.h>
#include <stdio.h>
#include "vtf3.h"

int main (void)
{
  const char *outfilename = "mytracefile";
  void *fcb;
  int writeunmergedrecord, writtenchars;
  size_t writtenbytes;

  (void) VTF3_InitTables ();
  fcb = VTF3_OpenFileOutput (outfilename, VTF3_FILEFORMAT_STD_ASCII,
                             writeunmergedrecord = 0);
  if (fcb == 0) {
    (void) printf ("Couldn't open %s\n", outfilename);
    return (127);
    }
  writtenchars = 8; /* file format header */
  writtenbytes = (size_t) writtenchars * sizeof (char);
  writtenchars = VTF3_WriteDefversion (fcb, VTF3_GetVersionNumber ());
  writtenbytes += (size_t) writtenchars * sizeof (char);
  writtenchars = VTF3_WriteDefcreator (fcb, "Stephan's TraceLib");
  writtenbytes += (size_t) writtenchars * sizeof (char);
  /* Write all the other records. */
  (void) VTF3_Close (fcb);
  (void) printf ("Wrote %lu bytes\n", (unsigned long) writtenbytes);
  return (0);
  }
```

Fig. 1. Trace file creator skeleton

A program skeleton to create VTFs might look as depicted in figure 1. *vtf3.h* is the header file which contains the VTF3 macros, types, prototypes and, last but not least, further application examples. To prevent unintentional interferences, all of the externally visible symbols of the implementing VTF3 libraries in themselves start with the substring `"VTF3_"`.

VTF3 needs to be initialized with respect to internal, statically allocated tables. To do so, one has to invoke *VTF3_InitTables()* as the first call to the API. This function can be executed multiple times, but, of course, only its first execution, which is not thread-safe, does some initialization. *VTF3_InitTables()* does not allocate any memory, it only sets up internal tables which are used by all the other interfaces in a read-only manner. The next is an open statement. The first argument to *VTF3_OpenFileOutput()* has to be a string which is used as a name for the output file. If the regular expression `".\.[gG][zZ]$"` matches this file name, then compressed output is generated, whereby the compression method is the same as one would get by the command *gzip(1)* called with `-1` or `--fast`, respectively. Sometimes, VTF3 does not have `zlib` support. To check whether this support is compiled in, the service function *VTF3_HaveZlib()* can be invoked, which returns `1` in the case that the VTF3-ZLIB-1.1.4 code is onboard, or `0`, otherwise.

In this regard, a comment for using VAMPIR follows. If the `Handled_by:`-line of the `File.Tracefile_Info` display shows that this VAMPIR comes with a VTF3 version with `zlib` support, one should disable the appropriate external converter with the help of the `Preferences.Tracefile.External_Converters` display. This does significantly improve the input performance. Because the built-in `zlib` code is not 64-bit-clean, the size of compressed files cannot exceed the 32-bit limitations. Files without compression do not have this problem.

The second argument to *VTF3_OpenFileOutput()* determines the trace file format. Possible values are *VTF3_FILEFORMAT_STD_ASCII*, *VTF3_FILEFORMAT_STD_BINARY* and *VTF3_FILEFORMAT_FST_ASCII*. The readable *Standard ASCII Format* is for debugging trace libraries etc. Upon creation, this format is the slowest one, but, with VTF3, its speed got acceptable, too. The *Standard Binary Format* is the fastest format which should be used for production. The *Fast ASCII Format* is an experimental format without restrictions. Even though the resulting files of this format are readable in a sense, their size may be smaller than that of appropriate binary files. Furthermore, it may be much faster than the *Standard ASCII Format*, because floating-point numbers are carried using the so-called XY-representation which reads the bit sequence of a floating-point number, after its translation into the IEEE-754 double-precision little-endian format, as a two-element array of unsigned 32-bit integers. All of the described trace file formats are accepted by VAMPIR.

If the third argument equals a non-zero value, then *VTF3_OpenFileOutput()* creates an *Unmerged* record after the file header. In that case, the file has to be post-processed, e.g. with the help of VPTMERGE. For files which should be ready for VAMPIR, the third argument is zero.

Upon successful completion, *VTF3_OpenFileOutput()* returns the address of a so-called *fcb* (file control block). `0` is returned to indicate an open error. Testing the *fcb* address against zero is an essential activity. The *fcb* itself has to be left untouched. Its address is passed to many other interfaces in order to get thread-safe, i.e. except for the first call of *VTF3_InitTables()*, the VTF3 API is thread-safe as long as not more than one thread works with a particular *fcb*. The maximum number of *fcb*s at the same time corresponds to the maximum number of open files, see also *getrlimit(2)*.

According to table 2, the first declaration record should be the *Defversion* record. This record defines the version of the trace file format definition that the VTF conforms to. VTF3 provides an appropriate number by the function *VTF3_GetVersionNumber()*. All of the record-writing routines return an *int* value containing the number of written characters. These numbers can be accumulated to ensure that the resulting files keep handy. For example, if the file size exceeds some limit without having written all the records, one should simply branch to the file close statement.

In figure 1, the next record provides creator information. This could be the name of any software, its version number, any date etc. If one does not have an idea how to do that, the return value of *VTF3_GetVersion()*, being a statically allocated string, can be plugged into the *Defversion* record interface.

After all the records have been written, the *fcb*-controlled device should be released calling the function *VTF3_Close()*. This flushes the buffers, closes the file and frees the memory. Normally, one *fcb* binds 2 MB of core space. This value can grow up to 20 MB, depending on the size of the largest record.

2.1 More Declaration Section Records

This section individually describes important declaration records. With respect to these declaration records, the situation is similar to the one in 'true' programming languages, it is intricate and wrapped in some mystery.

Defsyscpunums The *Defsyscpunums* record critically determines the layout of several Vampir displays, e.g. the global timeline. Exactly one record of this type has to exist. It defines the maximum degree of parallelism which can be visualized. From today's point of view, the interface looks somewhat strange, but, it has been defined in the past to handle Meta-Computers, too.

```
int VTF3_WriteDefsyscpunums (void *fcb,
                             int systemcpunumberarraydim,
                             const int *systemcpunumberarray);
```

Modern VTF3 applications put exactly one value onto the array `systemcpunumberarray`, being the maximum degree of parallelism. Hence, `systemcpunumberarraydim` should always equal 1. There are worthier records than this one to define the structure of a system of computers.

With more than about 50 parallel event streams, there are two alternative ways to go. First, for example, an application runs 1000 parallel processes and the position is that the trace file has to contain them all. Then, one would enter 1000 here and would have to claim the Vampir process filtering features as selector while visualizing. The second approach is that the trace library is an intelligent part of the particular application, being able to perform skilled selections by itself. One representative of this kind is FMC [3]. The trace library of FMC is an integral part which knows exactly what is going on. If the degree of parallelism exceeds a redefinable number, e.g. 40, then an activity-dependent selection policy is applied to find out the right candidates which are exclusively allowed to write traces. In the trace file, all of the others are mapped onto one dummy task which only takes up the messages between visible and invisible members. Hence, FMC would not call the *Defsyscpunums* interface with a value larger than 40. Finally, sequential applications invoke this interface with 1.

Defthreadnums A VTF may contain one *Defthreadnums* record. This record is useful if a multi-threaded MPI application is to be traced whereat messages are sent and received by threads which have to be represented as individual task objects, i.e. with an individual timeline each. For example, a message is sent from process 1 to process 0. Thread 1 of process 0 may actually receive this message because it was targeted on process 0, exacting on any thread of process 0. On the other hand, thread 1 of process 0 cannot obtain the correct sender, i.e. it can obtain the sender process, but not the sender thread. Provided that each thread has its own timeline, it is obvious that there is no way to write a correct trace record for the message above. For that reason, VAMPIR always draws message lines between processes, i.e. between the 0-thread timelines of these processes, even though this might be in apparent contradiction to some call stack information. Therefore, VAMPIR needs to know which thread belongs to which process, and this is exactly what the *Defthreadnums* record carries.

```
int VTF3_WriteDefthreadnums (void *fcb,
                             int threadnumarraydim,
                             const int *threadnumarray);
```

`threadnumarray` is an array of the dimension `threadnumarraydim`. The dimension signifies the number of true processes. The array elements contain the individual thread numbers for each process, in which `*(threadnumarray+0)` is the number of threads running in process 0, `*(threadnumarray+1)` is the number of threads running in process 1, and so on in appropriate order. The sum of all array elements has to equal the sum of all the array elements provided by the *Defsyscpunums* record. In the presence of the *Defthreadnums* record, VAMPIR uses all the task-indexing numbers in a special manner which is described below. Otherwise, VAMPIR takes these numbers as they are.

Defcpuname With restrictions, the default timeline task labeling can be influenced with the help of the VAMPIR `Preferences.General` display, i.e. with the `Process_Name` and `Process_Offset` fields there. Full timeline task labeling support is only provided by the VTF *Defcpuname* records. These records can be used to assign arbitrary application-dependent names to the tasks of the timelines. Especially if an intelligent trace library selects particular tasks for tracing by itself, the real application task numbers and the numbers from the default VAMPIR task numbering scheme cannot trivially be mapped onto each other. Hence, this kind of trace libraries should completely define their own task labeling. The appropriate VTF3 interface is

```
int VTF3_WriteDefcpuname (void *fcb,
                          unsigned int cpuid,
                          const char *cpuname);
```

With every call of the function *VTF3_WriteDefcpuname()*, one character string `cpuname` is passed, i.e. VAMPIR's default labeling of exactly one task can be overwritten whereat the particular task is selected by the unsigned integer `cpuid`.

The parameter `cpuid` is used for several VTF3 interfaces and needs to be explained. Traditionally, its value is one out of the interval

$$[\,0,\ \left(\ \sum_{i=0}^{i=\texttt{systemcpunumberarraydim}-1} \texttt{systemcpunumberarray}_i\ \right)\ -\ 1\,]\ , \quad (1)$$

which means that `cpuid`+1 may vary from 1 up to the maximum degree of parallelism, and `cpuid` is something like a plain task rank as it would be yielded by the MPI function *MPI_Comm_rank()* when called with *MPI_COMM_WORLD* as the communicator. But, the interval (1) only describes valid `cpuid` values in the absence of any *Defthreadnums* record. In the presence of a prior defined *Defthreadnums* record, the interpretation of `cpuid` totally changes, becoming somewhat sophisticated. In that case, VAMPIR reads the parameter `cpuid` as

$$\texttt{cpuid}\ =\ 2^{16}\times \textit{threadid}\ +\ \textit{processid}\ . \quad (2)$$

Applying (2), *processid* has to be any value out of the interval

$$[\,0,\ \texttt{threadnumarraydim}\ -\ 1\,]\ . \quad (3)$$

Furthermore, if k denotes the zero-based process index, possible *threadid* values would be such ones out of the interval

$$[\,0,\ \texttt{threadnumarray}_k\ -\ 1\,]\ . \quad (4)$$

Summarizing here, together with the *Defthreadnums* record, the semantics of the VTF3 parameter `cpuid` completely changes. Because of the applied thread index encoding scheme, in that case, the resulting ASCII output looks quite ugly, but, on the other hand, with this encoding scheme, it is easy to find the process a particular thread belongs to.

Defcpugrp Looking retrospectively at the VAMPIR history, several approaches for data grouping have been made. From today's point of view, the approach being associated with the *Defcpugrp* records is the most flexible one, and this is the only approach which should be applied now. In principle, a *Defcpugrp* record creates a named task group, subsuming one or more single tasks and/or one or more prior defined task groups. For example, with the help of these *Defcpugrp* records, trace file designers can bail the structure of the underlying machinery. The current VAMPIR reflects this structure when perambulating the `Filters.Processes` display. Furthermore, at least one *Defcpugrp* record is required to use the most recent records carrying performance counter information.

```
int VTF3_WriteDefcpugrp (void *fcb,
                         unsigned int cpugrpid,
                         int cpuorcpugrpidarraydim,
                         const unsigned int *cpuorcpugrpidarray,
                         const char *cpugrpname);
```

The parameter `cpugrpid` is a unique group token being used to refer to this group later. With respect to its value, the only restriction is that this value has to be one out of the interval

$$[\,2^{31},\ 2^{32}\ -\ 1\,]\,. \tag{5}$$

`cpuorcpugrpidarray` is an array of the dimension `cpuorcpugrpidarraydim`. The dimension signifies the number of members of the group. The array elements may contain *cpuid*s or prior declared *cpugrpid*s. The same members, i.e., *cpuid*s and/or *cpugrpid*s, may also be used to declare other groups. Consequently, there may be several trees beside each other, each represented by a group hierarchy, which look at the machinery from different points of view. `cpugrpname` is the name of the newly declared group.

Groups, which are defined to be used below in *Defsamp*/*Samp* records for task-related performance counter information, typically do not contain *cpugrpid*s, they only contain *cpuid*s, i.e. they are flat. Alongside, the size of these groups should not be too excessive to prevent performance degradations while drawing/redrawing the VAMPIR `Global_Displays.Counter_Timeline` display. On the other hand, groups defined to be used in *Defsamp*/*Samp* records for task-group-related performance counter information typically only contain *cpugrpid*s as their members. Nevertheless, if task-related performance counter information and task-group-related performance counter information should be juxtaposed inside the same counter timeline, a mixed case is conceivable and possible. In all of these cases, the order of the group members is significant, since VAMPIR directly derives the layout of appropriate displays from the group contents.

Initially, VAMPIR comes up with a default group for the task selector (see `Filters.Processes` display). If a trace file defines at least one group, then this default group is the group with the largest group token. Otherwise, if the trace file does not contain any group definition at all, VAMPIR uses the internally created group *Default* as the default group with all tasks selected.

Defsclfile If one is going to deposit source code location information (SCL), unique tokens `sclfiletoken` are to be defined, one after another, being associated with the source code file names `sclfilename`.

```
int VTF3_WriteDefsclfile (void *fcb,
                          int sclfiletoken,
                          const char *sclfilename);
```

The values for `sclfiletoken`s are not subjected to any restriction so that hash-table-based algorithms can be applied to derive unique tokens evaluating the file names. Furthermore, it is a good idea to use relative file name paths instead of absolute ones for `sclfilename` to keep a given project movable with respect to its top-level directory position.

During a VAMPIR session, source code files can only be popped up if the file name paths `sclfilename`, being plugged into the *Defsclfile* interface, are described relative to the path of the current working directory VAMPIR has been invoked from. At least until now, this is not a bug, this is a feature.

In this document, the term *Unique Token* means that a token is only unique with respect to other tokens of the same kind. Hence, a `sclfiletoken` here may be unique, even though there is a `scltoken` (next paragraph) with exactly the same value.

Defscl While the *Defsclfile* records above declare the SCL file tokens, the SCL tokens themselves are declared by the *Defscl* records here, using the following interface.

```
int VTF3_WriteDefscl (void *fcb,
                      int scltoken,
                      int sclarraydim,
                      const int *sclfiletokenarray,
                      const int *scllinepositionarray);
```

In principle, an SCL is a conglomeration of pairs, each consisting of a file name and a line position inside this file. The file name is represented by a prior declared file name token `sclfiletoken`. The line position with respect to this file stands for itself. `sclarraydim` is the number of these pairs, i.e. the dimension of the two input vectors. If k is assumed to be the zero-based pair index, then `*(sclfiletokenarray+`k`)` contains the file name token of the k-th pair, and `*(scllinepositionarray+`k`)` contains the appropriate line position. Entering of more than one pair is possible here to dominate, for example, call hierarchy situations with an invisible software layer between two visible ones, whereby invisible means that, e.g. the source code is not available, i.e. it cannot be instrumented. The source code viewer of VAMPIR allows one to switch between the different pairs, which are defined by one *Defsclfile* record, pressing the keys `U` or `D`, respectively. `U` stands for *Up*, and `D` stands for *Down*. With respect to the VTF3 interface, the level of the pair k is assumed to be lower than the level of the pair $k + 1$. `scltoken` denotes a unique SCL token being subjected to

$$\texttt{scltoken} > \mathit{VTF3\_SCLNONE}\ , \tag{6}$$

where the symbol *VTF3_SCLNONE* is defined by the header *vtf3.h*.

Defact A *Defact* record is a state group record, consisting of a unique state group token `activitytoken` and a state group name `activityname`. State group tokens are small integers with

$$\texttt{activitytoken} > \mathit{VTF3\_NOACT}\ . \tag{7}$$

State group tokens have to be small because VAMPIR directly uses them as indices for certain table-lookup operations, but the suppositionally sorted set of all of the defined state group tokens of a VTF does not necessarily have to be a dense integer number sequence. `activityname` is a name that the state group should be associated with. For example, the state group *Blas* could group all the `BLAS` routines, i.e. a state itself would be a member of this subroutine collection which is executed by the application. In VAMPIR, a state group is denoted by the term *Activity*. Another commonly used term is *State Class*.

```
int VTF3_WriteDefact (void *fcb,
                      int activitytoken,
                      const char *activityname);
```

Trace file designers should know that, for VAMPIR, *activities* are one of the most important selection categories. A lot of statistics displays are *activity*-based, and, the *activity*, i.e. the state group a state belongs to, decides on the color of a state, not the state itself. Perhaps VAMPIR 3.0 comes with predefined colors for the state group names *Application*, *Calculation*, *Communication*, *Flush*, *Highlight*, *I/O*, *Idle*, *MPI*, *MT*, *PVM*, *SHM* and *VT_API*, which, of course, have to be defined by the VTF, too, if necessary. For other, personally-defined state groups, the initial color is a randomly determined gray value being redefinable with the help of the `Preferences.Color_Styles.Activities` display. As with other changes, color modifications are also stored into files in the `${HOME}/.VAMPIR_defaults` directory. Hence, trace file designers will not always create new state group name entries, forcing the visualizer to redefine colors as the first action when examining a new trace file.

The pair `activitytoken` = *VTF3_NOACT* and `activityname` = `"NOACT"` is internally reserved and naturally defined, and its items cannot be used here. *VTF3_NOACT* or `"NOACT"`, respectively, is uncolored.

Defstate A *Defstate* record declares a state denoting a named program section which can be entered and left one or more times. Typically, states are subroutines or large loops which should be examined. The appropriate interface is

```
int VTF3_WriteDefstate (void *fcb,
                        int activitytoken,
                        int statetoken,
                        const char *statename,
                        int scltoken);
```

`activitytoken` refers to a prior defined *activity*, i.e. a prior defined state group, that the state should belong to. Again, `statetoken` is a unique state token which is used later to refer to this state. As the state group tokens, for the same reason, state tokens have to be small integers. In addition, state tokens are subjected to

$$\texttt{statetoken} > \mathit{VTF3\_NOSTATE}\ . \tag{8}$$

A state may not be a member of more than one state group, but, during a VAMPIR session, a state can transiently change into another, perhaps newly, i.e. transiently, created state group. In VAMPIR, a state is denoted by the term *Symbol*. `statename` is a name the state should be associated with. All of the strings which are plugged into VTF3 interfaces should not be too long; some VAMPIR display legends will look ugly otherwise. Especially for C++ programmers, this might be a problem because their names are quite large, though these names may be mangled. The state `statetoken` = *VTF3_NOSTATE*, `statename` = `"NOACT"`, belonging to the state group *VTF3_NOACT*, is internally reserved and defined, and none of the latter items can be used here. Finally, the majority of the users will invoke the *VTF3_WriteDefstate()* interface with *VTF3_SCLNONE* as argument to the last parameter `scltoken`.

Defsampclass *Sample*s below are temporal functions which have to be recorded and evaluated with the help of VAMPIR, too. Every *sample* exactly belongs to one named sample class. The *Defsampclass* record binds a unique sample class token `sampleclasstoken`, not being subjected to any other restriction, to a particular sample class name `sampleclassname`. The corresponding VTF3 interface looks as follows

```
int VTF3_WriteDefsampclass (void *fcb,
                            int sampleclasstoken,
                            const char *sampleclassname);
```

FMC [3], for example, uses one of the strings `"Instructions"`, `"L1Dcache"`, `"L1Icache"`, `"L2Cache"`, `"TLB"`, `"DTLB"` and `"ITLB"` as sample class name, depending on a particular command line argument.

Defsamp To specify more properties, associated with *sample*s below, there is the following VTF3 interface.

```
int VTF3_WriteDefsamp (void *fcb,
                       int sampletoken,
                       int sampleclasstoken,
                       int iscpugrpsamp,
                       unsigned int cpuorcpugrpid,
                       int valuetype,
                       const void *valuebounds,
                       int dodifferentiation,
                       int datarephint,
                       const char *samplename,
                       const char *sampleunit);
```

`sampletoken` has to be a unique sample definition token, not subjected to further restrictions. It is used to shorten later references to the set of properties which is described here. `sampleclasstoken` is the token of the sample class that this sample description belongs to. If the argument `iscpugrpsamp` differs from zero, then the following argument `cpuorcpugrpid` is enforced to be a task group, i.e. the argument `cpuorcpugrpid` is OR-ed with the task group bit *VTF3_-CPUGRP_MASK* before it is evaluated. This behavior is somewhat strange and is a heritage of the past. To avoid confusion, always pass 1 to `iscpugrpsamp` and pass a prior defined *cpugrpid* group token to the `cpuorcpugrpid` argument, even if a single task application has to be traced. For the contents of the group here, it is a good idea to read once again the comments on the *Defcpugrp* record on page 9.

*Sample*s carry 64-bit unsigned integers or 64-bit floating-point data, whereby, depending on the result of a run-time analysis, IEEE-754 double precision or native CRAY floating-point numbers are accepted by the interfaces. Appropriately, one of the constants, *VTF3_VALUETYPE_UINT* or *VTF3_VALUETYPE_FLOAT*, has to be passed to `valuetype`, and the argument `valuebounds` might be read as the address of an imaginary union `valuebounds`

```
union valuebounds {
  struct {
    uint64_t   umin;
    uint64_t   umax;
    } ubounds ;
  struct {
    float64_t  fmin;
    float64_t  fmax;
    } fbounds ;
  } *valuebounds;
```

with certain minimum and maximum values filled into the right structure. If these boundary values do not exist or if they are unknown, the constants

`valuebounds->ubounds.umin` = 0 and
`valuebounds->ubounds.umax` = (*uint64_t*)~(*uint64_t*)0, or

`valuebounds->fbounds.fmin` = `-1.0e+300` and
`valuebounds->fbounds.fmax` = `+1.0e+300`,

respectively, can be entered. All the VTF3 library routines replace floating-point mantissae, being less than 10^{-300}, by zero, and replace such ones being greater than 10^{+300} by the value 10^{+300}, even if on CRAYs. Furthermore, IEEE-754 double-precision patterns `+NaN` and `+INF`, for example, are supposed to be valid floating-point numbers, which are larger than 10^{+300}. All clipping operations of this kind are prepared by bit manipulations so that floating-point exceptions should never occur. Internally, the VTF3 library routines do not deal with 64-bit integers. For strict X3J11 C89 conformance, eight-element character arrays and two-element arrays of at least 32-bit integers are used instead, together with a mechanism for endianess detection at run-time.

The argument `dodifferentiation` is used to decide whether the temporal function, which is given as a set of points (*sample*s), should interval-wise be differentiated during the VAMPIR session (`dodifferentiation!=0`) or should be shown as it is (`dodifferentiation==0`). For hardware performance counters, any value `dodifferentiation!=0` is recommended.

The argument `datarephint` has to be initialized with one of the constants *VTF3_DATAREPHINT_BEFORE*, *VTF3_DATAREPHINT_POINT*, *VTF3_-DATAREPHINT_AFTER*, or *VTF3_DATAREPHINT_SAMPLE*. At the moment, *VTF3_DATAREPHINT_BEFORE* is the only supported hint for VAMPIR. Once implemented, the differences between *VTF3_DATAREPHINT_BEFORE* and the other values will be subtle and should be tested. *VTF3_DATAREP-HINT_BEFORE* is the right constant for hardware performance counter recording, now and in the future.

The argument `samplename` is the address of a character string which determines the name of the temporal function, and `sampleunit` determines its physical unit. To enforce consistency at all, the physical unit might also directly be appended to the `samplename` string, separated by a times sign or a slash character, whereas the `sampleunit` string is left empty, i.e., the latter string only consists of a `'\0'`-character. If `dodifferentiation` above differs from zero, the unit string, or the function string, respectively, need to be lengthened by `"/s"`. The function will not be understood otherwise.

FMC [3], for example, uses the strings `"FloatingPointInstructions/s"`, `"Instructions/s"`, `"NormalizedL1DcacheMisses"`, `"L1DcacheMisses/s"`, `"L1DcacheAccesses/s"`, `"NormalizedL1IcacheMisses"`, `"L1IcacheMisses-/s"`, `"L1IcacheAccesses/s"`, `"NormalizedL2CacheMisses"`, `"L2CacheMisses-/s"`, `"L2CacheAccesses/s"`, `"TLBMisses/s"`, `"NormalizedDTLBMisses"`, `"D-TLBMisses/s"`, `"DTLBAccesses/s"`, `"NormalizedITLBMisses"`, `"ITLBMisses-/s"` and `"ITLBAccesses/s"` as sample names, depending on a particular command line argument. If a sample name starts with the substring `"Normalized"`, then the appropriate temporal function is an artificial one, ranging between 0 and 1, or a so-called derived counter, computed by the FMC on-board trace library. It should be noted that some processors do not allow one to measure *Misses/s* and *{Hits/Accesses}/s* at the same time. When detected, the FMC trace library, for example, switches back to a *Misses/s*–only measurement case. If there is more than one *sample* definition in a trace file, VAMPIR initially comes up with the one being switched on, which has the smallest `sampletoken` value.

Defcommunicator *MPI*, for example, supports so-called collective communication operations. The group of participating processes is defined by their communicator arguments. VAMPIR should also know the task members (*cpuid*s) of the communicators in use. The appropriate VTF3 interface is

```
int VTF3_WriteDefcommunicator (void *fcb,
                               int communicator,
                               int communicatorsize,
                               int tripletarraydim,
                               const unsigned int *tripletarray);
```

The parameter `communicator` denotes a unique token being subjected to

$$\texttt{communicator} \; != \; \mathit{VTF3\_NOCOMMUNICATOR}\,.$$

`communicatorsize` signifies the number of tasks (*cpuid*s) belonging to this communicator which cannot vary here. With respect to the parameter `triplet-`

`array`, the address of an imaginary structure array `cbounds` could be plugged in as argument.

```
struct {
  unsigned int   cllim;
  unsigned int   culim;
  unsigned int   cstep;
  } cbounds [tripletarraydim];
```

This structure array would have to be initialized in such a way that the the following code fragment correctly prints. Even though the type of `*tripletarray`, i.e. the type of the inner loop controlling variables, is unsigned, trace file designers should have plenty of room to define their communicator members efficiently.

```
int i;
for (i = 0; i < tripletarraydim; i++) {
  unsigned int cpuid;
  for (cpuid  = cbounds[i].cllim;
       cpuid <= cbounds[i].culim;
       cpuid += cbounds[i].cstep)
    (void) printf ("Cpuid %u is member of the communicator\n", cpuid);
  }
```

Fig. 2. Illustration of the communicator member definition

Defmsgname The *Defmsgname* interface allows the trace file designer to attach a name string to a token pair consisting of a message type (tag) and a communicator token. By default, when displaying a message, Vampir only shows this pair numerically.

```
int VTF3_WriteDefmsgname (void *fcb,
                          int msgtype,
                          int communicator,
                          const char *msgname);
```

In a message display, an additional name only appears if there is an appropriate *Defmsgname* entry with the same message type and the same communicator token, in which the special communicator token

`communicator` == *VTF3_NOCOMMUNICATOR* .

matches for all communicator tokens. Fortran-based applications may use the MPI *MPI_Comm_f2c()* function to transfer Fortran communicator handles to C communicator handles, which can directly be plugged in the *Defmsgname* interface in many cases.

Defglobalop Collective communication operations may have a name attached to them. The appropriate interface is the following one.

```
int VTF3_WriteDefglobalop (void *fcb,
                           int globaloptoken,
                           const char *globalopname);
```

globaloptoken has to be a unique collective communication operation token, not subjected to further restrictions. It is used to shorten later references to the globalopname name string argument. Typical values to globalopname are "MPI_Barrier" and "AllToAll".

Defiofile To trace basic I/O or MPI I/O, used file names have to be associated with unique file name tokens iofiletoken. These tokens are not subjected to any restriction so that hash-table-based algorithms can be applied to derive unique tokens evaluating the names. In displays, Vampir only shows the last file name component, i.e. it cuts off all the leading components looking for usual component separators.

```
int VTF3_WriteDefiofile (void *fcb,
                         int iofiletoken,
                         int communicator,
                         const char *iofilename);
```

For basic I/O, communicator has to be set to *VTF3_NOCOMMUNICATOR*, whereas in case of MPI I/O, communicator should be the communicator token being associated with the *MPI_File_Open()* function call. This is the only way Vampir can distinguish between basic and MPI I/O. Useful values to iofilename are "output.dat", "shome:myfile" and "Host23:File12", where the leading substring, up to the colon character, might be the abbreviated name of a node. The strings are not at all necessarily the same ones as they are used to open the file.

Defclkperiod Vampir needs a unit that all the unsigned-integer-like timestamp multipliers, carried by the event section records, have to be multiplied by. This unit is called clock period here, and it is the reciprocal number of ticks per second of any crystal-controlled counter, or it is a virtual quantity only.

```
int VTF3_WriteDefclkperiod (void *fcb,
                            double clkperiod);
```

Due to a VTF3 implementation restriction, the argument to clkperiod should not be too small, ensuring that the timestamp multipliers never exceed the value 10^{30}. On the other hand, clkperiod should not be too large to assure a sufficient resolution, i.e. to ensure that the quantization effects, caused by the fact that the timestamp multipliers are rounded to the next integer, keep neglectable. In our days, for typical performance analysis purposes, arguments to clkperiod range between 10^{-10} and 10^{-6}.

Deftimeoffset In the absence of a *Deftimeoffset* record, which is the normal situation in performance analysis, VAMPIR displays relative times according to

$$vampirtime = timestampmultiplier \times \texttt{clkperiod} \,. \tag{9}$$

This behavior completely changes if a *Deftimeoffset* record exists. Then, formula (10) is applied instead of formula (9),

$$\begin{aligned} vampirtime = {} & timestampmultiplier \times \texttt{clkperiod} \\ & + \texttt{timeoffset} \\ & + \text{"}\texttt{Thu Jan 1 00:00:00 1970}\text{"} \,, \end{aligned} \tag{10}$$

and VAMPIR displays absolute times at all. The *Deftimeoffset* record is useful when accounting data of a parallel machine or other fiscal information have to be visualized with the help of VAMPIR, for example. Typically, the function *time(2)* is used in that case, together with the *(double)* cast operator.

```
int VTF3_WriteDeftimeoffset (void *fcb,
                             double timeoffset);
```

For historical reasons, even though the appropriate parameter is of the type *double*, the argument to `timeoffset` is rounded to the next integer value and has to be out of the interval $[0, 2^{32} - 1]$. The unit of `timeoffset` is seconds.

Remaining Declaration Section Records The remaining declaration section records *Defsyscpunames*, *Defclstr*, *Defact_obsol*, *Defstate_obsol*, *Defcpuregclass*, *Defclstrregclass*, *Defcpureg*, *Defclstrreg*, *Defredfunc_obsol*, *Defkparreg*, *Defpatternshape* and *Defpattern* are not described in this document. Most of them are outmoded; the latter three touch commercial or research matters.

2.2 Event Section Records

This section individually describes important event section records. All of the records of the event section carry a timestamp, or more accurately, they carry at least one unsigned-integer-like timestamp multiplier that the clock period is multiplied by to get a correct time value inside VAMPIR. From now on, for short, these unsigned-integer-like timestamp multipliers are simply called time. And, once more, event section records have to be chronologically sorted, finally.

Table 3 shows the currently known VTF3 event section record types. Record types tagged with (†) are outmoded. The one with the (•) should not be written by the user. It has been defined to internally handle unrecognizable records or such ones which obviously carry bad data.

Table 3. The event section record types

1. *Comment*	9. *Mutexrelease*	17. *Globalop*
2. *Downto*	10. *Samp*	18. *Kparregbegin*
3. *Upfrom*	11. *Cpuregval* (†)	19. *Kparregbarsum*
4. *Upto*	12. *Clstrregval* (†)	20. *Kparregend*
5. *Exchange* (†)	13. *Fileiobegin*	21. *Pattern*
6. *Exchange_obsol* (†)	14. *Fileioend*	22. *Unrecognizable* (•)
7. *Srcinfo_obsol* (†)	15. *Sendmsg*	
8. *Mutexacquire*	16. *Recvmsg*	

Comment In general, comment records do not have any meaning for VAMPIR. They have been introduced in the past to explain the trace files themselves. Their time value ensures that the relative position of comments does not change while sorting operations. With VAMPIR 3.0 [2], some comments got a special meaning. Those ones, having a zero time and starting with the substring `"ENV "` at the first position, are presented in the `File.Tracefile_Info` display, too, where the substring `"ENV "` is removed from the top.

```
int VTF3_WriteComment (void *fcb,
                       double time,
                       const char *comment);
```

With respect to the event record interfaces, the times which go in here are of the type *double*. The `'\0'`-terminated string that `comment` has to point to will not be stripped, i.e. whitespace characters will be left as the are.

Downto, Upfrom, Upto State exchange records are the most important records at all. The appropriate interfaces are the following ones.

```
int VTF3_WriteDownto (void *fcb,
                      double time,
                      int statetoken,
                      unsigned int cpuid,
                      int scltoken);
int VTF3_WriteUpfrom (void *fcb,
                      double time,
                      int statetoken,
                      unsigned int cpuid,
                      int scltoken);
  int VTF3_WriteUpto (void *fcb,
                      double time,
                      int statetoken,
                      unsigned int cpuid,
                      int scltoken);
```

VAMPIR assumes that all the tasks of a parallel program, i.e. the timelines which have to be visualized, initially start from the reserved state *VTF3_NOSTATE*.

A *Downto* record flags that at the time `time` (page 17), the state of the task `cpuid` (page 8) downwards changes to a new state `statetoken` (page 11). This typically happens when a function is entered, i.e. the call stack dives into the next lower level. `scltoken` denotes a prior defined source code location (SCL) token (page 10), or *VTF3_SCLNONE*, if SCLs are not used.

Both the *Upfrom* and the *Upto* records perform the complementary operation. They are applied to flag state exchanges in an upwards direction. *Upfrom* expects the token of the state being left as the `statetoken` argument, whereas *Upto* assumes that `statetoken` contains the token of the target state. While tracing, in most cases, the use of *Upfrom* records is simpler, since there is no need to maintain a call/return stack at all. Nevertheless, for programs which execute a *longjmp(3)* function, for example, some extra trace code is necessary to ensure that the up/down state exchanges always keep balanced. Furthermore, it is a common failure to forget to maintain the call/return stack during run-time phases of a program, in which the tracing is temporarily switched off. Finally, proper programs finish all of the traces returning to the state *VTF3_NOSTATE*, i.e. with an empty call/return stack for every `cpuid`.

If Vampir is used to visualize system accounting information, the *activityname* (page 10), or the *activitytoken* (page 10), respectively, can be associated with any project ID string, and the *statename* (page 11), or the `statetoken`, respectively, can be associated with any job ID string. Another scenario could be to associate the *activitytoken* with a user ID string and the `statetoken` with a job ID string.

Mutexacquire, Mutexrelease The following two interfaces support MUTual EXclusion device tracing, for example, to trace multi-threaded codes using the *pthread_mutex_lock(3)* and *pthread_mutex_unlock(3)* functions.

```
int VTF3_WriteMutexacquire (void *fcb,
                            double time,
                            unsigned int cpuid,
                            int enterstatetoken,
                            int leavestatetoken,
                            int leavestatetokenisupfrom,
                            double durationtimesteps,
                            int mutexsize,
                            const void *mutex,
                            int scltoken);
int VTF3_WriteMutexrelease (void *fcb,
                            double time,
                            unsigned int cpuid,
                            int enterstatetoken,
                            int leavestatetoken,
                            int leavestatetokenisupfrom,
                            double durationtimesteps,
                            int mutexsize,
                            const void *mutex,
                            int scltoken);
```

They have been defined to visualize whether the flow of an application is significantly disturbed by critical section execution. Each, the *Mutexacquire* and the *Mutexrelease* record, cause a complete *Downto*–{*Upfrom/Upto*} state exchange operation, depending on the argument `leavestatetokenisupfrom`. If `leavestatetokenisupfrom` differs from zero, this state exchange operation is of the type *Downto–Upfrom*, or is of the type *Downto–Upto*, otherwise. Accordingly, if `leavestatetokenisupfrom` differs from zero, `leavestatetoken` equals `enterstatetoken`, or, if not, `leavestatetoken` has to be the state token that the operation starts from. For the *Mutexacquire* record, `enterstatetoken` might be associated with the function *pthread_mutex_lock()*, and, for the *Mutexrelease* record, `enterstatetoken` might be understood as the token for *pthread_mutex_unlock()*. Again, `cpuid` denotes the task, executing the lock/unlock operation. The argument `durationtimesteps`, being a particular unsigned-integer-like timestamp multiplier value, too, describes the time being spent in *pthread_mutex_lock()* or *pthread_mutex_unlock()*, respectively. `mutex` might be the address of the mutex variable and `mutexsize` its size. Using Pthreads, something special has to be taken into account. The specification does not say anything about the type *pthread_mutex_t*. Hence, *pthread_mutex_t* might be implemented as a structure, with or without gaps not necessarily being initialized. Another point is that the contents of a mutex may vary during its life-time. To avoid appropriate problems, i.e., to ensure that `mutex` is always correctly recognized, mutexes should be tokenized and the tokens should be used instead of the variables themselves. Useful token strings could be created from any unified virtual address, which points to the particular mutex object, lengthened by the IPV4/IPV6 node address. Mutex tokenization is quite uncomfortable, of course, but it is safe. VAMPIR does not do more than checking `mutex`es against each other for equality.

Samp The interface described here is used to carry the values of one or more temporal functions, being called *sample*s, as these values have been picked up at the same time `time` each.

```
int VTF3_WriteSamp (void *fcb,
                    double time,
                    unsigned int cpuorcpugrpid,
                    int samplearraydim,
                    const int *sampletokenarray,
                    const int *samplevaluetypearray,
                    const void *samplevaluearray);
```

The argument `samplearraydim` contains the number of different *sample*s this record carries. Since every carried *sample* may have its own definition (page 12), the particular sample tokens are passed through the integer array `sampletokenarray`. On the other hand, each *Defsamp* record above is associated with one group (argument *cpuorcpugrpid* on page 12) that this definition is only valid for. Furthermore, the *Samp* record argument `cpuorcpugrpid` here needs to be a member of such a group. Hence, since there is only one possible value `cpuorcpugrpid` in a *Samp* record, only such *sample*s can be carried together, which belong to a *Defsamp* definition with a *cpuorcpugrpid* value, containing this *Samp* record argument `cpuorcpugrpid` as a member.

The values, which have to be loaded onto the integer array `samplevalue-typearray`, have to be the same as the ones, one has specified with the *Defsamp* records belonging to the appropriate elements of `sampletokenarray`, i.e. the *sample* types have to match. Accordingly, with respect to the parameter `sample-valuearray`, the address of an imaginary union array `samplevalue` could be plugged in as argument, having loaded the data either onto the unsigned integer type members, or onto the floating-point type ones.

```
union {
   uint64_t  usamplevalue;
   float64_t fsamplevalue;
   } samplevalue [samplearraydim];
```

Typically, for hardware performance counters, the *Samp* argument `cpuorcpu-grpid` is a particular *cpuid* belonging to a flat group (page 9). `samplearraydim` may range between 2, being the valid minimum for all known processors with hardware performance counters, and 7, being the value for HITACHI's SR8000. Furthermore, all of the elements of the `samplevaluetypearray` are typically set to the constant *VTF3_VALUETYPE_UINT*, and, the array `samplevaluearray` is filled with data of the type *uint64_t*.

Fileiobegin, Fileioend The following two interfaces for basic I/O or MPI I/O work in pairs. *Fileiobegin* records the beginning of an I/O operation, *Fileioend* records its end. Accordingly, the `time` argument to *Fileiobegin* has to be the time when functions as *fread (3)* or *fwrite (3)*, for example, are called and the `time` argument to *Fileioend* has to be the one when these functions finish. File open and close operations do not play a role here; they are handled with the help of the state exchange records above.

```
int VTF3_WriteFileiobegin (void *fcb,
                           double time,
                           unsigned int cpuid,
                           int fileiotype,
                           int iofiletoken,
                           int bytescopied,
                           int scltoken);

  int VTF3_WriteFileioend (void *fcb,
                           double time,
                           unsigned int cpuid,
                           int fileiotype,
                           int iofiletoken,
                           int bytescopied,
                           int scltoken);
```

The arguments `cpuid`, `fileiotype`, `iofiletoken` and `bytescopied` are exactly the same for one *Fileiobegin/Fileioend* pair belonging to the same I/O operation, VAMPIR ignores all mismatching records of this type. With respect to the `cpuid` argument, proper trace libraries always use the correct value. On the other hand, lazy implementations may only use the *processid* instead (see eqn. (2) at page 8), i.e. the *cpuid* value belonging to the zero thread of the same process.

Possible values for `fileiotype` are *VTF3_FILEIOTYPE_READ* and *VTF3_-FILEIOTYPE_WRITE*. The argument to `iofiletoken` is a prior defined token (page 16). `bytescopied` denotes the number of bytes being read/written from/onto the I/O medium. Formatted I/O, as it is typically applied for the standard I/O streams, needs some extra code to determine the number of transmitted bytes, and this code does not have anything to do with the design of these trace records here.

Sendmsg, Recvmsg The *Sendmsg* and *Recvmsg* interfaces have been designed to trace basic send/receive operations of applications using the message-passing paradigm. Mainly, MPI has influenced the rich set of interface parameters, but, of course, PVM and CRAY SHMEM are well supported, too.

```
int VTF3_Sendmsg (void *fcb,
                  double time,
                  unsigned int sender,
                  unsigned int receiver,
                  int communicator,
                  int msgtype,
                  int msglength,
                  int scltoken);

int VTF3_Recvmsg (void *fcb,
                  double time,
                  unsigned int receiver,
                  unsigned int sender,
                  int communicator,
                  int msgtype,
                  int msglength,
                  int scltoken);
```

Sendmsg records the beginning of a message transfer, *Recvmsg* records its end. Accordingly, the `time` argument to *Sendmsg* has to be the time when a function as *MPI_Send()*, for example, is called, and the `time` argument to *Recvmsg* has to be the one when *MPI_Recv()*, for example, finishes.

The arguments `sender`, `receiver`, `communicator` and `msgtype` are exactly the same for one *Sendmsg/Recvmsg* pair belonging to the same message transfer operation. With respect to the `sender/receiver` arguments, proper trace libraries always use the correct values. On the other hand, lazy implementations may only use the *processid*s instead (see eqn. (2) at page 8), i.e. *sender/receiver* values belonging to the zero thread of the appropriate processes.

The `communicator` argument may have been defined (page 14) or not. Vampir simply takes `communicator` as a magic number, which has to match while looking for the appropriate *Recvmsg* record entry, examining all the records of the `receiver` stream, which normally carry a larger `time` value than the current *Sendmsg* record entry. The argument `msgtype` denotes a message type (tag). It may or may not occur in a prior defined *Defmsgname* record (page 15). `msglength` is the number of bytes being transmitted, which does not need to be the same at the sender and the receiver side.

MPI, for example, allows one to use so-called wildcard operands to describe some data. A similar mechanism is not implemented with Vampir. Hence, all the data going into these interfaces have to be resolved properly, which needs some extra code.

Finally, it is necessary to have synchronized clocks for all the tasks, being involved in a message-passing program, i.e. the resulting offsets between all these clocks do not disturb as long as they do not exceed the minimal message transmission time. Oftentimes, the latter condition is not fulfilled at cluster-based systems, and the amount of extra code, working around this situation, is important. For example, FMC [3] uses a general software approach, which takes one minute execution time at all, to locally determine the two parameters for a linear hardware clock correction with sufficient precision, and it should be noted that, because of the tree algorithms, it is not a problem having 10^3 CPUs on the run.

Globalop The *Globalop* interface is for collective communication operations as they are commonly used with MPI, for example.

```
int VTF3_Globalop (void *fcb,
                   double time,
                   int globaloptoken,
                   unsigned int cpuid,
                   int communicator,
                   unsigned int rootcpuid,
                   int bytessent,
                   int bytesreceived,
                   double durationtimesteps,
                   int scltoken);
```

The `time` argument describes the local time the collective communication operation started. The argument `durationtimesteps`, being an unsigned-integer-like timestamp multiplier value, too, describes the local time being spent in the collective operation. `globaloptoken` is a prior defined collective communication operation token (page 16), which is associated with the name of the collective communication operation.

Again, with respect to the `cpuid/rootcpuid` arguments, proper trace libraries always use the correct values. On the other hand, lazy implementations may only use the *processid*s instead (see eqn. (2) at page 8), i.e. *cpuid/rootcpuid* values belonging to the zero thread of the appropriate processes.

While other records may have a `communicator` argument which is defined or not, the *Globalop* interface always needs a defined one (page 14). What at this moment keeps is a question one has to ask here. What should the contents of `communicator` in case of multi-threaded MPI be, for example, the correct `cpuid` values (see eqn. (2) at page 8), if used, or the appropriate *processid*s only. The current practice is that communicators only contain *processid*s, i.e. the *cpuid* values belonging to the zero thread of the appropriate processes.

Furthermore, if a particular collective communication operation does not define a *rootcpuid* parameter, an invalid value has to be used to flag that. With respect to further VAMPIR developments, the following value seems to be safe

$$\mathit{invalidcpuid} = \left(\sum_{i=0}^{i=\mathtt{systemcpunumberarraydim}-1} \mathtt{systemcpunumberarray}_i \right), \quad (11)$$

which, probably, will never be used for anything else (ref. to page 6 and eqn. (1) at page 8). To explain the parameters `bytessent` and `bytesreceived`, an example should be discussed. We assume that the processes 0, 1, 2 and 3 perform an *MPI_Bcast()* operation, in which 0 is the `rootcpuid`, broadcasting 4 bytes. Then, the `bytessent` argument belonging to process 0 is 4, clearly, and the `bytesreceived` arguments of all the others are 4, too. The `bytesreceived` argument for the `rootcpuid` 0 is also 4, because *MPI_Bcast()* also sends to itself, theoretically, at least. The non-`rootcpuid` processes do not send anything, and this does not mean that they send 0 bytes here. MPI allows one to send or receive 0 bytes. If a process or thread does not send or receive anything, then the appropriate arguments have to be negative; otherwise, inside VAMPIR, the interpretation with respect to the character of a particular collective communication operation may fail.

Remaining Event Section Records The remaining event section records *Exchange*, *Exchange_obsol*, *Srcinfo_obsol*, *Cpuregval*, *Clstrregval*, *Kparregbegin*, *Kparregbarsum*, *Kparregend* and *Pattern* are not described in this document. Most of them are outmoded, some touch commercial or research matters.

3 Controlling and Service Interfaces

The following interface has already been explained on page 4.

void `VTF3_InitTables` (*void*);

The following interface has already been explained on page 4.

```
void *VTF3_OpenFileOutput (const char *outputfilename,
                           int outputfileformat,
                           int writeunmergedrecord);
```

If the third argument equals a non-zero value, then *VTF3_OpenFileOutput()* creates an *Unmerged* record after the file header. Another way to create this record is to let the `writeunmergedrecord` argument be zero and call the following function instead, i.e. the *Unmerged* record can also be written by hand. Keep in mind that the *Unmerged* record has to be the absolutely first one of a file.

```
int VTF3_WriteDefunmerged (void *fcb);
```

Instead of opening the record output to write it onto a file, the record output can also be opened to process it inside the memory. The appropriate interface looks as follows.

```
void *VTF3_OpenMemoryOutput (int outputfileformat);
```

Again, possible values for the argument `outputfileformat` are *VTF3_FILEFORMAT_STD_ASCII*, *VTF3_FILEFORMAT_STD_BINARY* and *VTF3_FILEFORMAT_FST_ASCII*. *VTF3_OpenMemoryOutput()* returns the address of an *fcb* (file control block). The *fcb* itself has to be left untouched. If opened with *VTF3_OpenMemoryOutput()* instead of *VTF3_OpenFileOutput()*, the records cannot be written with the

```
int VTF3_Write... (void *fcb, ...);
```

interfaces, they have to be created with the help of appropriate 'composers'

```
VTF3_rec_t *VTF3_Compose... (void *fcb, ...); .
```

Composers expect the same arguments as the 'writers'. On the other hand, instead of returning the number of written characters, composers return the address of a so-called *rcb* (record control block). The *rcb* itself belongs to the VTF3 API and should only be read. The returned *rcb* address keeps valid as long as no further composer is invoked with the same *fcb*. The type of the *rcb* is *VTF3_rec_t* which is defined in *vtf3.h*.

```
typedef const struct {
  int         type;
  int         numchars;
  const char  *record;
  } VTF3_rec_t;
```

```
#include <stdio.h>
#include <stddef.h>
#include "vtf3.h"

int main (void)
{
  const char *outfilename = "mytracefile";
  const unsigned char formatheaders [] [8] = VTF3_HEADER_ALL_INIT;
  int fileformat = VTF3_FILEFORMAT_STD_ASCII;
  FILE *fp;
  size_t writtenbytes;
  void *fcb;
  VTF3_rec_t *rcbaddress;

  fp = fopen (outfilename, "wb");
  if (fp == 0) {
    (void) printf ("Couldn't open %s\n", outfilename);
    return (127);
    }
  (void) fwrite (&formatheaders[fileformat][0], sizeof (char), 8, fp);
  writtenbytes = 8 * sizeof (char); /* file format header */
  (void) VTF3_InitTables ();
  fcb = VTF3_OpenMemoryOutput (fileformat);
  rcbaddress = VTF3_ComposeDefversion (fcb, VTF3_GetVersionNumber ());
  if (fileformat == VTF3_FILEFORMAT_STD_BINARY)
      (void) fwrite (rcbaddress->record, sizeof (char),
                     (size_t) rcbaddress->numchars, fp);
    else
      (void) fprintf (fp, "%s\n", rcbaddress->record);
  writtenbytes += (size_t) rcbaddress->numchars * sizeof (char);
  rcbaddress = VTF3_ComposeDefcreator (fcb, "Stephan's TraceLib");
  if (fileformat == VTF3_FILEFORMAT_STD_BINARY)
      (void) fwrite (rcbaddress->record, sizeof (char),
                     (size_t) rcbaddress->numchars, fp);
    else
      (void) fprintf (fp, "%s\n", rcbaddress->record);
  writtenbytes += (size_t) rcbaddress->numchars * sizeof (char);
  /* Compose and write all the other records. */
  (void) VTF3_Close (fcb);
  (void) fclose (fp);
  (void) printf ("Wrote %lu bytes\n", (unsigned long) writtenbytes);
  return (0);
  }
```

Fig. 3. Trace file creator skeleton to understand the composers

The structure member `type` is a magic number which is associated with the record type. These numbers are defined by the macros beginning with the substring `"VTF3_RECTYPE_"` in *vtf3.h*. Because they occur in thousands of existing binary trace files, these magic numbers have never been changed. The meaning of the other members gets clear with the help of the program skeleton in fig. 3, which exactly produces the same output as the example in fig. 1 at page 4.
In binary format, the structure member `record` is the complete record. In both ASCII formats, the new-line delimiter `^J`, which is normally part of the ASCII records, too, is replaced by a terminating `'\0'`-character, i.e. the new-line delimiter has to be written by hand in fig. 3.
The following two interfaces have been designed for Vptmerge. *VTF3_GetDefRecTypeArrayDim()* returns the number of existing declaration section records,

while the *Unmerged* record is not taken into account. VPTMERGE evaluates this value to allocate an integer array which is used as the argument for *VTF3_GetDefRecTypeArray()*. When *VTF3_GetDefRecTypeArray()* is back, this integer array contains the record type magic numbers of the declaration section records in a particular order so that VPTMERGE can obtain information on how to sort them. Hence, the sort order of the declaration section records, being produced by VPTMERGE, is determined by the VTF3 code, not by VPTMERGE.

int `VTF3_GetDefRecTypeArrayDim` (*void*);

void `VTF3_GetDefRecTypeArray` (*int* `*defrecordtypes`);

The function *VTF3_GetRecTypeArrayDim()* returns the number of existing records in total. Applications evaluate this value to allocate memory which is used in connection with some other VTF3 interfaces.

int `VTF3_GetRecTypeArrayDim` (*void*);

VTF3_GetRecTypeArray() stores the record type magic numbers of all existing records onto the integer array, `recordtypes` is pointing to. The user has to accept that these numbers are stored in any unknown order.

void `VTF3_GetRecTypeArray` (*int* `*recordtypes`);

VTF3_GetCopyHandlerArray() stores the *VTF3_Write...*-function addresses of all existing records onto the array, `copyhandlers` is pointing to. The user can be sure, that, for each *k*, `*(recordtypes+`*k*`)` and `*(copyhandlers+`*k*`)` belong to the same record.

void `VTF3_GetCopyHandlerArray` (*VTF3_handler_t* `*copyhandlers`);

VTF3_DebugHandler() invokes the *exit(3)* system function with the argument 127 after the string `"VTF3: VTF3_DebugHandler() invoked\n"` has been printed. Take note that the type of *VTF3_DebugHandler()* matches *VTF3_handler_t*.

int `VTF3_DebugHandler` (*void* `*firsthandlerarg, `...);

The next is an open function for reading from a file. The first argument to *VTF3_OpenFileInput()* has to be a string which is used as name for the input file. If the regular expression `".\.[gG][zZ]$"` matches this file name, then, while reading, the file contents is passed through an uncompressor filter, using `zlib`, before it is processed. Of course, this can only happen, if `zlib` support is compiled in. Furthermore, it should be noted that `zlib` switches into the transparent mode, if it does not understand the input. Because the built-in `zlib` code is not 64-bit-clean, the size of compressed files cannot exceed the 32-bit limitations. Files without compression do not have this problem.

```
void *VTF3_OpenFileInput (const char *inputfilename,
                          const VTF3_handler_t *handlers,
                          void * const *firsthandlerargs,
                          int substitudeupfrom);
```

The second argument to *VTF3_OpenFileInput()* is an array of pointers to so-called record handler functions. This needs some background. While later reading the file, its contents is completely scanned and parsed by certain VTF3 routines to assemble record-related argument lists, which are passed to appropriate functions of the user, being the handlers for the particular record types. The record-related argument lists are assumed to be the same ones as for the record writers *VTF3_Write...* . As with the *VTF3_Write...*-functions, the user record handlers have to return an integer value which is checked to be non-negative. A negative value from a user record handler terminates the whole application. One difference exists. While the *VTF3_Write...*-functions always wait for a valid *fcb* as the first argument, associated with an output file, the user record handlers here may have individual first arguments which are explained later.

The user has to define a handler for every existing record, i.e. the array, `handlers` is pointing to, should be large enough to store *VTF3_GetRecTypeArrayDim()* elements of the type *VTF3_handler_t*. The place, inside this array, where the handler for the record with a particular record type magic number has to be stored, is the one with the same index that this record type magic number occupies in `recordtypes` above. More simply, it is required that for each k, `*(recordtypes+`k`)` and `*(handlers+`k`)` belong to the same record.

In practice, four different cases are of interest. First, a particular handler function pointer is initialized with the value *(VTF3_handler_t) 0*. Then, while later reading, all the records of this type are silently ignored. Secondly, a handler function pointer is initialized with the value *VTF3_DebugHandler* (page 27). Then, while later reading, the function *VTF3_DebugHandler()* will not be invoked. A message is printed instead, which says that the occurrence of the appropriate record would lead to a *VTF3_DebugHandler()* call, before the application is terminated. Hence, the function *VTF3_DebugHandler()* only exists to have a valid address, which some VTF3 routines can test against. Third, a particular record handler function pointer is initialized with the appropriate *VTF3_Write...*-function, which needs the *(VTF3_handler_t)* cast operator. Then, while later reading, all records of this type are copied onto an opened file. From now on, it should be clear why the function *VTF3_GetCopyHandlerArray()* (page 27), which stores all the *VTF3_Write...*-function addresses onto an array, has its name. From the point of view of the record handlers, the *VTF3_Write...*-functions are nothing else than copying handlers, copy handlers for short. The last case is the one, a handler function pointer is initialized with the address of a user-defined function.

The third argument to *VTF3_OpenFileInput()* is an array of *(void *)*-pointers. It has to be initialized so that the k-th record handler function can be invoked with `*(firsthandlerargs+`k`)` as the first argument. For user-defined functions, `*(firsthandlerargs+`k`)` typically represents the address of a communication control block which may or may not contain *fcb*-addresses. Even if the array element `*(handlers+`k`)` has been initialized with *(VTF3_handler_t)0*, `*(firsthandlerargs+`k`)` should get a value, *(void *)0*, for example.

If the last argument, `substitudeupfrom`, differs from zero, then, while reading, all *Upfrom*-records (page 18) are replaced by appropriate *Upto*-records. Vampir itself always opens the input devices with `substitudeupfrom=1`. All other applications should open them with `substitudeupfrom=0`.

Upon successful completion, *VTF3_OpenFileInput()* returns the address of an *fcb*. `0` is returned to indicate an open error. Testing the *fcb* address against zero is an essential activity.

Two more functions might be interesting here, *VTF3_WriteUnrecognizable()* and *VTF3_ComposeUnrecognizable()*.

```
int VTF3_WriteUnrecognizable (void *fcb,
                              double lastvalidtime,
                              int numberofunrecognizablechars,
                              int typeofunrecognizablerecord,
                              const char *unrecognizablerecord);
```

```
VTF3_rec_t *VTF3_ComposeUnrecognizable (void *fcb,
                              double lastvalidtime,
                              int numberofunrecognizablechars,
                              int typeofunrecognizablerecord,
                              const char *unrecognizablerecord);
```

Of course, nobody would use them to create trace file records, but taken as handlers for erroneous input parts, they do a good job. An error-flagging *Comment* record is produced, being helpful for debugging. `lastvalidtime` is the time value of the last correct event section record, or 0, if there was none before. `numberofunrecognizablechars` is the dimension of the bad area with the start address `unrecognizablerecord`. If the type of the damaged record could be determined, it is given by `typeofunrecognizablerecord`, otherwise, the parameter has the value *VTF3_RECTYPE_UNRECOGNIZABLE*.

VTF3_ReadFileInput() is one of the functions which process the input after opening with *VTF3_OpenFileInput()*. Once invoked, the user does not get back the control up to the end of the file. The input is scanned and parsed to call the user-defined record handler functions, one after another. *VTF3_ReadFileInput()* returns the number of read bytes.

```
size_t VTF3_ReadFileInput (void *fcb);
```

VTF3_ReadFileInputLtdBytes() is similar to *VTF3_ReadFileInput()*. Here, the amount of bytes, which is read by one call, can be limited with the help of the argument `bytestoberead`. The returned value, the number of read bytes in effect, is only less than the value of this argument, if the end of the file has been reached. In case of reading compressed input, there is a problem. For some reasons, `bytestoberead` is internally rounded up to 2 MB, approximately.

```
size_t VTF3_ReadFileInputLtdBytes (void *fcb,
                                   size_t bytestoberead);
```

VTF3_ReadFileInputLtdRecs() allows one to limit the number of records which are read by one call. The returned value, the number of read records in effect, is never larger than the value of the argument `recordstoberead`. The returned value may be less than `recordstoberead`, if the end of the file has been reached. In any order, all functions, *VTF3_ReadFileInput()*, *VTF3_ReadFileInputLtdBytes()* and *VTF3_ReadFileInputLtdRecs()*, can be used to process the same input stream.

```
int VTF3_ReadFileInputLtdRecs (void *fcb,
                               int recordstoberead);
```

VTF3_OpenMemoryInput() has been designed to read trace files, or parts of them, respectively, from the memory. This function is similar to *VTF3_OpenFileInput()* (page 27). The first argument is not a file name, it is a file format instead. Possible values are *VTF3_FILEFORMAT_STD_ASCII*, *VTF3_FILEFORMAT_STD_BINARY* and *VTF3_FILEFORMAT_FST_ASCII*. The returned *fcb* is always valid.

```
void *VTF3_OpenMemoryInput (int inputfileformat,
                            const VTF3_handler_t *handlers,
                            void * const *firsthandlerargs,
                            int substitudeupfrom);
```

VTF3_ReadMemoryInput() belongs to *VTF3_OpenMemoryInput()*. The file control block address *fcb* is the one, *VTF3_OpenMemoryInput()* has returned. Once invoked, beginning at `src`, *VTF3_ReadMemoryInput()* reads `size` bytes and processes them assuming the trace file format `inputfileformat`, being the first argument to *VTF3_OpenMemoryInput()*. `src` may contain zero, one or more trace records. *VTF3_ReadMemoryInput()* can be executed multiple times.

```
size_t VTF3_ReadMemoryInput (void *fcb,
                             const void *src,
                             size_t size);
```

QueryFormat() is for all kinds of *fcb*-controlled (opened) devices. It returns the currently used trace file format. *QueryFormat()* always returns *VTF3_FILEFORMAT_STD_ASCII*, *VTF3_FILEFORMAT_STD_BINARY* or *VTF3_FILEFORMAT_FST_ASCII*. Because of the following circumstances, this might be a problem. The *Legacy ASCII Format*, being no longer created, does not have a file format header on top. If there is not a known file format header on top of a file, then this file is processed by the *Standard ASCII Format* parser, operating in legacy mode. Hence, to internally invoke the *Standard ASCII Format* parser, *VTF3_FILEFORMAT_STD_ASCII* is returned by *QueryFormat()* if the format cannot be recognized finding a valid header. This is a bad point. Trying to read the executable */bin/sh* as a trace file, for example, this cannot be avoided, but, fortunately, the *Standard ASCII Format* parser is strong enough to dominate in such a situation.

int `VTF3_QueryFormat` (*const void* `*fcb`);

VTF3_Close() is the function to close an *fcb*-controlled device. It flushes the buffers, closes the file, if one has been opened, and frees the memory.

void `VTF3_Close` (*void* `*fcb`);

Putting all the discussed open, write/compose/read and close functions into table 4, their dependencies on each other should be clear.

Table 4. Matching open, write/compose/read and close functions

Open	Matching Write/Compose/Read	Close
VTF3_OpenFileOutput()	*VTF3_Write...()*	*VTF3_Close()*
VTF3_OpenMemoryOutput()	*VTF3_Compose...()*	*VTF3_Close()*

VTF3_OpenFileInput()	*VTF3_ReadFileInput()*
	VTF3_ReadFileInputLtdBytes()
	VTF3_ReadFileInputLtdRecs()
		VTF3_Close()

VTF3_OpenMemoryInput()	*VTF3_ReadMemoryInput()*	*VTF3_Close()*

Sometimes, VTF3 does not have `zlib` support. To check whether this support is compiled in, the service function *VTF3_HaveZlib()* can be invoked, which returns `1` in case that the VTF3-ZLIB-1.1.4 code is on-board, or `0`, otherwise. *VTF3_HaveZlib()* is the only function which can be used without a prior call of *VTF3_InitTables()*.

int `VTF3_HaveZlib` (*void*);

VTF3_GetVersion() returns a pointer to a statically allocated string which describes the VTF3 version. It is derived from the static `rcsid` string at run-time. VTF3-1.37 with VTF3-ZLIB-1.1.4, for example, yields `"TUD/ZHR vtf3 1.37 2003/11/25 18:23:15 with zlib 1.1.4"`.

const char `*VTF3_GetVersion` (*void*);

VTF3_GetVersionNumber() returns an integer value which describes the VTF3 version number. It is derived from the static `rcsid` string at run-time. VTF3-1.37, for example, yields 30010037.

int `VTF3_GetVersionNumber` (*void*);

4 Application Examples

The first application example implements a commonly used function with the name *QueryFormat()*, returning one of the VTF3 file format magic numbers for the not necessarily open file, `filename` is pointing to. It uses the VTF3 interface *VTF3_QueryFormat()* which always returns *VTF3_FILEFORMAT_STD_ASCII*, *VTF3_FILEFORMAT_STD_BINARY* or *VTF3_FILEFORMAT_FST_ASCII*. In addition, *QueryFormat()* may also return *VTF3_FILEFORMAT_UNDEFINED*, if the file with `filename` cannot be opened.

```
/* Begin of Application Example 1. */

#include <stdlib.h>
#include <stddef.h>
#include <string.h>
#include "vtf3.h"

int QueryFormat (const char *filename)
{
  int nrectypes, substitudeupfrom, fileformat;
  VTF3_handler_t *handlers;
  void **firsthandlerargs, *fcb;

  /* Initialize VTF3. */
  (void) VTF3_InitTables ();

  /* How many different record types do exist ? */
  nrectypes = VTF3_GetRecTypeArrayDim ();

  /* Allocate two auxiliary arrays, one for the user record
     handler function pointers and one for the first arguments to
     the user record handler functions. */
  handlers = (VTF3_handler_t *) malloc ((size_t) nrectypes *
                                         sizeof (VTF3_handler_t));
  firsthandlerargs = (void **) malloc ((size_t) nrectypes *
                                        sizeof (void *));
  if (handlers == 0 || firsthandlerargs == 0) {
    (void) printf ("No more memory\n");
    (void) exit (127);
    }

  /* We shall not read anything, but, ‘VTF3_OpenFileInput()’
     will copy both arrays into dark places.
     Make Purify and Valgrind happy. */
  (void) memset (handlers, 0,
                 (size_t) nrectypes * sizeof (VTF3_handler_t));
  (void) memset (firsthandlerargs, 0,
                 (size_t) nrectypes * sizeof (void *));
  /* Open the input device. */
  fcb = VTF3_OpenFileInput (filename, handlers, firsthandlerargs,
                            substitudeupfrom = 0);

  /* Check the returned file control block address. */
  if (fcb != 0) {
      /* It is valid, ask for the file format. */
      fileformat = VTF3_QueryFormat (fcb);
      /* Close the input device. */
      (void) VTF3_Close (fcb);
      }
    else
      /* The file could not be opened,
         the file format keeps undefined. */
      fileformat = VTF3_FILEFORMAT_UNDEFINED;

  /* Free the auxiliary vectors. */
  (void) free (firsthandlerargs);
  (void) free (handlers);

  /* Give back the determined value. */
  return (fileformat);
  }

/* End of Application Example 1. */
```

The second application example is an artificial one. It shows a lot of VTF3 features. The reader should test himself. The imaginary job which has to be done here is the following one. Read a trace file of any known format in portions of 50000 bytes, approximately, and translate this file into the *Standard Binary Format*. Any found *Defunmerged* records have to be ignored. Furthermore, in all *Sendmsg* records, *cpuid* 12 should be replaced by *cpuid* 14. Additionally, the first *Sendmsg* record with the global maximum value for *msglength* has to be printed onto *stdout*, in ASCII format, of course. Any found *Srcinfo_obsol* records should abort the execution. Finally, if any unrecognizable record is found in the input file, this record has to be processed by the standard *Unrecognizable* record handler, which translates it into an error-flagging *Comment* record. Moreover, if the input file is of the *Standard ASCII Format*, each unrecognizable record has to be passed to an extern 'alien' ASCII record parser.

```
/* Begin of Application Example 2. */

#include <stddef.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include "vtf3.h"
```

```
/* Define a communication control block type for later use.
   It will be passed to some user installed record handlers. */
typedef struct {
  void *fcboutfile;
  void *fcboutmemory;
  void *fcbin;
  int  maxmsglength;
  char *sendmsgrecord;
  } ccb_t;

/* Two true user record handlers have to be installed here,
   one for the Sendmsg records and one for the Unrecognizable
   records. Declare their prototypes.
   Let the compiler perform strict type checking. */
VTF3_DCL_SENDMSG        (static, MySendmsgHandler       );
VTF3_DCL_UNRECOGNIZABLE (static, MyUnrecognizableHandler);

/* Declare the extern ‘alien’ ASCII record parser. */
extern int MyAlienParser (void *anydata, int numchars,
                           const char *record);

/*****************************************************************/

/* Now define the first user handler for the Sendmsg records.
   Basically, all record handlers have to give back a
   non-negative integer, otherwise, the processing will be
   aborted after return. */
static int MySendmsgHandler (void *ccbvoid, double time,
                              unsigned int sender,
                              unsigned int receiver,
                              int communicator, int msgtype,
                              int msglength, int scltoken)
{
  ccb_t *ccb;
  int writtenbytes;
  VTF3_rec_t *record;

  /* Prevent excessive casting. */
  ccb = (ccb_t *) ccbvoid;
```

```
/* Check for ‘cpuid’ translation.
   Keep in mind, ‘sender’ and ‘receiver’ are located in the
   stack area. Therefore, they can be modified without any
   problem. In general, it is not allowed to destroy any data
   coming in here. */
if (sender == 12)
  sender = 14;
if (receiver == 12)
  receiver = 14;

/* Invoke the predefined copy handler to put the modified record
   onto the binary output stream. Remember, all the predefined
   copy handlers want to see the control block of an opened
   output device. */
writtenbytes = VTF3_WriteSendmsg (ccb->fcboutfile, time, sender,
                                  receiver, communicator,
                                  msgtype, msglength,
                                  scltoken);

/* Now check the message lengths. */
if (ccb->sendmsgrecord != 0 && msglength <= ccb->maxmsglength)
  /* There is stored any Sendmsg record with a larger or equal
     ‘msglength’ value than this record has.
     We are ready to go out from here. */
  return (writtenbytes);

/* O.k., we have to store this Sendmsg record in ASCII format.
   VTF3_ComposeSendmsg() is used as the record assembler.
   VTF3_ComposeSendmsg() also needs an opened output device,
   just for the buffers. But, at least because of the different
   formats, this output device cannot be the same as for
   VTF3_WriteSendmsg().
   Hence, another output device has to be used. */
record = VTF3_ComposeSendmsg (ccb->fcboutmemory, time, sender,
                              receiver, communicator, msgtype,
                              msglength, scltoken);

/* Really store now.
   Pay attention, the memory of data, coming in here from VTF3,
   is recycled after return. Therefore, one has to save all the
   things which should survive. */
ccb->maxmsglength = msglength;
if (ccb->sendmsgrecord != 0)
  (void) free (ccb->sendmsgrecord);
  /* Pay attention, because of the file format
     which will be specified later, ‘record->record’ is
     a ’\0’-terminated printable ASCII string. */
  ccb->sendmsgrecord = (char *) malloc ((strlen (record->record)
                                         + 1) * sizeof (char));
  if (ccb->sendmsgrecord == 0) {
    (void) printf ("No more memory\n");
    (void) exit (127);
    }
  (void) strcpy (ccb->sendmsgrecord, record->record);

  /* Ready to leave this function. */
  return (writtenbytes);
  }

/*****************************************************************/

/* Now define the second user handler
   for all Unrecognizable records. */
static int MyUnrecognizableHandler (void *ccbvoid,
                                    double lastvalidtime,
                                    int numberofunrecognizablechars,
                                    int typeofunrecognizablerecord,
                                    const char *unrecognizablerecord)
{
  ccb_t *ccb;
  int writtenbytes, rc;

  /* Prevent excessive casting. */
  ccb = (ccb_t *) ccbvoid;

  /* Firstly, invoke the predefined copy handler which puts an
     appropriate error-flagging comment record onto the binary
     output stream. Remember, all the predefined copy handlers
     want to see the control block of an opened output device. */
  writtenbytes = VTF3_WriteUnrecognizable (ccb->fcboutfile,
                                           lastvalidtime,
                                           numberofunrecognizablechars,
                                           typeofunrecognizablerecord,
                                           unrecognizablerecord);
  /* Pass the Unrecognizable record to the ‘alien’ ASCII record
     parser, if the input file is of the Standard ASCII Format. */
  if (VTF3_QueryFormat(ccb->fcbin) == VTF3_FILEFORMAT_STD_ASCII) {
    /* Pay attention, ‘unrecognizablerecord’
       is not ’\0’-terminated. */
    rc = MyAlienParser ((void *) 0, numberofunrecognizablechars,
                        unrecognizablerecord);
    if (rc < 0)
      /* This would abort the trace file processing. */
      writtenbytes = rc;
    }

  /* Ready to leave this function. */
  return (writtenbytes);
  }

/*****************************************************************/

/* Now define the main program. */
int main (int argc, char **argv)
{
  ccb_t ccb;
  int writeunmergedrecord, nrectypes, *recordtypes, i,
    substitudeupfrom;
  VTF3_handler_t *handlers;
  void **firsthandlerargs;
  size_t bytesread, bytestoberead;

  /* Initialize VTF3. */
  (void) VTF3_InitTables ();

  /* Open a binary output device. */
  ccb.fcboutfile = VTF3_OpenFileOutput ("binfile",
                                        VTF3_FILEFORMAT_STD_BINARY,
                                        writeunmergedrecord = 0);
  if (ccb.fcboutfile == 0) {
    (void) printf ("Couldn’t open %s\n", "binfile");
    return (127);
    }

  /* Open an ASCII in-core output device. */
  ccb.fcboutmemory =
    VTF3_OpenMemoryOutput (VTF3_FILEFORMAT_STD_ASCII);
```

```
/* Continue the communication control block initialization. */
ccb.maxmsglength  = 0;
ccb.sendmsgrecord = 0;

/* Again, how many different record types do exist ? */
nrectypes = VTF3_GetRecTypeArrayDim ();

/* Allocate three auxiliary arrays, one for record type magic
   numbers, one for the record handler function pointers
   and one for the first arguments to the record handler
   functions. */
recordtypes = (int *) malloc ((size_t) nrectypes *
                              sizeof (int));
handlers = (VTF3_handler_t *) malloc ((size_t) nrectypes *
                                      sizeof (VTF3_handler_t));
firsthandlerargs = (void **) malloc ((size_t) nrectypes *
                                     sizeof (void *));
if (recordtypes == 0 || handlers == 0 || firsthandlerargs == 0){
  (void) printf ("No more memory\n");
  return (127);
  }

/* Store the record type magic numbers onto the appropriate
   array. Pay attention, the caller does not know their ordering
   scheme. */
(void) VTF3_GetRecTypeArray (recordtypes);

/* Store the predefined copy handler function pointers onto my
   array. */
(void) VTF3_GetCopyHandlerArray (handlers);

/* What follows, this is the final handler table setup. */
for (i = 0; i < nrectypes; i++) {

  if (recordtypes[i] == VTF3_RECTYPE_DEFUNMERGED) {
    /* Defunmerged records have to be suppressed. Install 0 as
       the handler. Resetting ‘firsthandlerargs[i]’ makes Purify
       and Valgrind happy. */
    handlers[i] = 0;
    firsthandlerargs[i] = 0;
    continue;
    }
if (recordtypes[i] == VTF3_RECTYPE_SENDMSG) {
  /* Replace the predefined copy handler by our own one,
     do not forget to redirect the first argument, too. */
  handlers[i] = (VTF3_handler_t) MySendmsgHandler;
  firsthandlerargs[i] = &ccb;
  continue;
  }

if (recordtypes[i] == VTF3_RECTYPE_SRCINFO_OBSOL) {
  /* To check whether anybody has created a trace file
     containing Srcinfo_obsol records in the past, install
     the builtin debug handler. This handler will never be
     invoked, it only represents a valid machine address which
     can internally be compared to. Resetting
     'firsthandlerargs[i]' makes Purify and Valgrind happy. */
  handlers[i] = VTF3_DebugHandler;
  firsthandlerargs[i] = 0;
  continue;
  }

if (recordtypes[i] == VTF3_RECTYPE_UNRECOGNIZABLE) {
  /* Replace the predefined copy handler by our own one, do
     not forget to redirect the first argument, too. Pay
     attention, since 'ccb.fcbin' is not known up to now, it
     keeps uninitialized.
     At the moment, this is not a problem. */
  handlers[i] = (VTF3_handler_t) MyUnrecognizableHandler;
  firsthandlerargs[i] = &ccb;
  continue;
  }

/* All the other handlers keep the predefined copy handlers,
   directly seeing the opened output device control block. */
firsthandlerargs[i] = ccb.fcboutfile;

/* End of loop. */
}
```

```
  /* Open the input device, correctly installing the handlers.
     Furthermore, a value for ‘ccb.fcbin’ is returned so
     that ‘ccb’ initialization can be completed, being late but
     early enough with respect to the handlers. */
  ccb.fcbin = VTF3_OpenFileInput ("anyfile", handlers,
                                  firsthandlerargs,
                                  substitudeupfrom = 0);
  if (ccb.fcbin == 0) {
    (void) printf ("Couldn’t open %s\n", "anyfile");
    return (127);
    }

  /* Free the auxiliary arrays. */
  (void) free (firsthandlerargs);
  (void) free (handlers);
  (void) free (recordtypes);

  /* Now push the operation
     to portion-wise process the input file contents. */
  do bytesread = VTF3_ReadFileInputLtdBytes (ccb.fcbin,
                                          bytestoberead = 50000);
    while (bytesread != 0);

  /* Print now, if something is to be done. */
  if (ccb.sendmsgrecord != 0) {
    (void) printf ("%s\n", ccb.sendmsgrecord);
    (void) free (ccb.sendmsgrecord);
    }

  /* Close all devices. */
  (void) VTF3_Close (ccb.fcbin);
  (void) VTF3_Close (ccb.fcboutmemory);
  (void) VTF3_Close (ccb.fcboutfile);

  return (0);
  }

/* End of Application Example 2. */
```

A final comment. As long as it is clear what VAMPIR wants to see, writing trace files is easy, reading and processing them, too. Nevertheless, the second application example should be understood as a case to study how to go. It seems that it is a good idea to extract the code from this document to perform personal tests. Good luck.

5 Trademark Acknowledgement

All trademarks are property of their respective trademark owners.

References

1. Nagel W.E., Arnold A., Weber M., Hoppe H-C., and Solchenbach, K.: *VAMPIR: Visualization and Analysis of MPI Resources.* Supercomputer 63, Vol. 12, No. 1, pp. 69-80, 1996.
2. Pallas GmbH: *VAMPIR 3.* http://www.pallas.com
3. Seidl S., Nagel W.E., and Brunst.H.: *The Future of HPC at SGI: Early Experience with SGI SN-1.* in Proceedings of the Sixth European SGI/Cray MPP Workshop (Sep 7-8, 2000, Manchester, UK), B.J.Jesson, ed., pp. 209-219, 2000.