GASPI: Global Address Space Programming Interface

Specification of a general purpose API for communication

draft, version 0.91

Daniel Grünewald et al.
Fraunhofer ITWM, Fraunhofer Platz 1, 67663 Kaiserslautern

September 19, 2012

Contents

1 About the document **7**

2 Introduction to GASPI **7**

2.1 Overview and Goals 7

2.2 History 7

2.2.1 Application requirements 8

2.2.2 Design goals 8

3 GASPI terms and conventions **8**

3.1 Naming Conventions 8

3.2 Procedure specification 9

3.3 Semantic terms 9

3.4 Examples 10

4 GASPI concepts **10**

4.1 Introduction and overview 10

4.2 GASPI processes 11

4.3 GASPI groups 11

4.4 GASPI segments 11

4.5 GASPI one-sided communication 11

4.6 GASPI passive communication 12

4.7 GASPI global atomics 13

4.8 GASPI collective communication 13

4.9 GASPI timeouts 14

4.10 GASPI return values 14

5 GASPI definitions **15**

5.1 Types 15

5.2 Constants 17

6 Execution model **17**

6.1 Introduction and overview 17

6.2 Process configuration 19

6.2.1 GASPI configuration structure 19

6.2.2 gaspi_config_get 21
6.2.3 gaspi_config_set 21
6.3 Process management calls 22
6.3.1 gaspi_proc_init 22
6.3.2 gaspi_proc_num 24
6.3.3 gaspi_proc_rank 24
6.3.4 gaspi_proc_term 25
6.3.5 gaspi_proc_kill 25
6.3.6 Example 26
6.4 Connection management utilities 28
6.4.1 gaspi_connect 28
6.4.2 gaspi_disconnect 29
6.5 State vector for individual processes 30
6.5.1 Introduction 30
6.5.2 gaspi_state_vec_get 31
6.6 MPI Interoperability 32
6.7 Argument checks and performance 33

7 Groups 33
7.1 Introduction 33
7.2 Gaspi group generics 34
7.2.1 Gaspi group type 34
7.2.2 GASPI_GROUP_ALL 34
7.3 Group creation 34
7.3.1 gaspi_group_create 34
7.3.2 gaspi_group_add 35
7.3.3 gaspi_group_commit 35
7.4 Group deletion 36
7.4.1 gaspi_group_delete 36
7.5 Group utilities 37
7.5.1 gaspi_group_num 37
7.5.2 gaspi_group_size 38
7.5.3 gaspi_group_ranks 38

8 Gaspi segments 39

8.1 Introduction and overview 39
8.2 Segment creation 40
8.2.1 `gaspi_segment_alloc` 40
8.2.2 `gaspi_segment_register` 41
8.2.3 `gaspi_segment_create` 42
8.3 Segment deletion 44
8.3.1 `gaspi_segment_delete` 44
8.4 Segment utilities 44
8.4.1 `gaspi_segment_num` 44
8.4.2 `gaspi_segment_ptr` 45
8.5 Segment memory management 46

9 One-sided communication 47
9.1 Introduction and overview 47
9.2 Basic communication calls 48
9.2.1 `gaspi_write` 48
9.2.2 `gaspi_read` 50
9.2.3 `gaspi_wait` 52
9.2.4 Examples 53
9.3 Weak synchronisation primitives 57
9.3.1 Introduction 57
9.3.2 `gaspi_notify` 58
9.3.3 `gaspi_notify_waitsome` 59
9.3.4 `gaspi_notify_reset` 61
9.4 Extended communication calls 62
9.4.1 `gaspi_write_notify` 62
9.4.2 `gaspi_write_list` 64
9.4.3 `gaspi_write_list_notify` 65
9.4.4 `gaspi_read_list` 66
9.5 Communication utilities 67
9.5.1 `gaspi_queue_size` 67
9.5.2 `gaspi_queue_purge` 68

10 Passive communication 68
10.1 Introduction and overview 68

10.2 Passive communication calls 69
10.2.1 `gaspi_passive_send` 69
10.2.2 `gaspi_passive_wait` 71
10.2.3 `gaspi_passive_receive` 72
10.3 Passive communication utilities 74
10.3.1 `gaspi_passive_queue_size` 74
10.3.2 `gaspi_passive_queue_purge` 75

11 Global atomics 76
11.1 Introduction and Overview 76
11.2 Atomic operation calls 76
11.2.1 `gaspi_counter_set` 76
11.2.2 `gaspi_counter_fetch_add` 77
11.2.3 `gaspi_counter_compare_swap` 78
11.2.4 Examples 79

12 Collective communication 82
12.1 Introduction and overview 82
12.2 Barrier synchronisation 83
12.2.1 `gaspi_barrier` 83
12.2.2 Examples 84
12.3 Predefined global reduction operations 85
12.3.1 `gaspi_allreduce` 85
12.3.2 Predefined reduction operations 86
12.3.3 Predefined types 87
12.4 User-defined global reduction operations 87
12.4.1 `gaspi_allreduce_user` 87
12.4.2 User defined reduction operations 88
12.4.3 allreduce state 89

13 GASPI getter functions 89
13.1 Getter functions for group management 90
13.1.1 `gaspi_group_max` 90
13.2 Getter functions for segment management 90
13.2.1 `gaspi_segment_max` 90

13.3 Getter functions for communication management 91
13.3.1 `gaspi_queue_num` 91
13.3.2 `gaspi_queue_size_max` 91
13.3.3 `gaspi_transfer_size_max` 91
13.3.4 `gaspi_notification_num` 92
13.4 Getter functions for passive communication 92
13.4.1 `gaspi_passive_queue_size_max` 92
13.4.2 `gaspi_passive_transfer_size_max` 93
13.5 Getter functions for atomic operations 93
13.5.1 `gaspi_counter_num` 93
13.6 Getter functions for collective communication 93
13.6.1 `gaspi_allreduce_buf_size` 93
13.6.2 `gaspi_allreduce_elem_max` 94
13.7 Getter functions related to infrastructure 94
13.7.1 `gaspi_network_type` 94
13.7.2 `gaspi_build_infrastructure` 95

14 GASPI Environmental Management 95
14.1 Implementation Information 95
14.1.1 `gaspi_version` 95
14.2 Timing information 96
14.2.1 `gaspi_time_get` 96
14.2.2 `gaspi_time_ticks` 96
14.3 Error Codes and Classes 97
14.3.1 GASPI error codes 97
14.3.2 `gaspi_error_message` 97

15 Profiling Interface 98
15.1 Statistics 98
15.1.1 `gaspi_statistic_counter_get` 98
15.1.2 `gaspi_statistic_counter_reset` 99
15.1.3 `gaspi_statistic_counter_collect` 99

A Listings 100
A.1 success_or_die 100

A.2 wait_if_queue_full 101

1 About the document

This is a draft version for comment only. There are parts of a complete specification missing, for example you will find (nearly) nothing about language bindings. On the other hand, there could be very well too much redundancy at some places. However, most of the ideas for a new PGAS API are included in this draft version and it should give you a good impression of how the final version will look like.

2 Introduction to GASPI

2.1 Overview and Goals

GASPI is a Partitioned Global Address Space (PGAS) API. It aims at scalable, flexible and failure tolerant computing in parallel computing environments. GASPI targets extreme scalability and aims to initiate a paradigm shift from bulk-synchronous two-sided communication patterns towards an asynchronous communication and execution model. To that end GASPI leverages one-sided RDMA driven communication in a Partitioned Global Address Space. In contrast to other efforts in the PGAS community, GASPI is neither a new language (like e. g. Chapel from Cray), nor an extension to a language (like e. g. Co-Array Fortran). Instead—very much in the spirit of MPI—it complements existing languages like C/C++ or Fortran with a PGAS API which enables the application to leverage the concept of the Partitioned Global Adress Space. In contrast to e. g. OpenShmem or Global Arrays, GASPI is not limited to a single memory model, but rather provides configurable RDMA pinnend PGAS memory segments. By means of these segments GASPI does not merely support a single Partitioned Global Address Space, but rather a multitude of Partitioned Global Address Spaces. In GASPI it is hence quite possible for different memory management systems—or indeed different PGAS applications—to co-exist in the same single global address space. GASPI is failure tolerant in the sense that it provides timeout mechanisms for all non-local procedures, failure detection and the possibility to adapt to shrinking or growing node sets.

2.2 History

The GASPI specification originates from the PGAS API of the Fraunhofer ITWM (Fraunhofer Virtual Machine, FVM), which has been developed since 2005. Starting from 2007 this PGAS API has evolved into a robust commercial product (called GPI) which is used in the industry projects of the Fraunhofer ITWM. GPI offers a highly efficient and scalable programming model for Partitioned Global Address Spaces and has replaced MPI completely at Fraunhofer ITWM. In 2011 the partners of Fraunhofer ITWM, Fraunhofer SCAI, TUD, T-Systems SfR, DLR, KIT, FZJ, DWD and Scapos have initiated and launched the GASPI project to define a novel specification for a PGAS API (GASPI, based on GPI) and to make this novel GASPI specification a reliable, scalable and universal tool for the HPC community.

2.2.1 Application requirements

The following application requirements have been identified

- one-sided asynchronous remote read/write operations
- allreduce (also for subgroups)
- barrier (also for subgroups)
- broadcast (also for subgroups)
- passive communication
- atomic counters

2.2.2 Design goals

From the application requirements the following design goals have been deduced:

- Extreme scalability
- Multi-segment support (heterogeneous systems, NUMA-pinning)
- Dynamic allocation of segments
- Timeout mechanisms and failure tolerance
- Group support for collectives
- Large flexibility in the number of queues, queue sizes, atomic counters etc.
- A maximum freedom to implementors, where details are left to implementation
- A strong standard library which takes care of convenience procedures and cosmetics. The specification should be simple and solid.
- GPI is the base and we should improve from there where necessary

3 GASPI terms and conventions

This section describes notational terms and conventions used throughout the GASPI document.

3.1 Naming Conventions

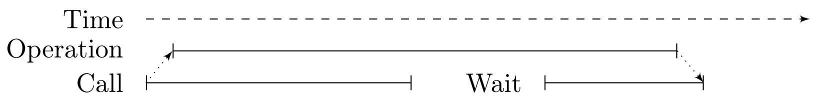
All procedures are named in accordance with the following convention. The procedures have `gaspi_` as a prefix. The prefix is followed by the operation name.

3.2 Procedure specification

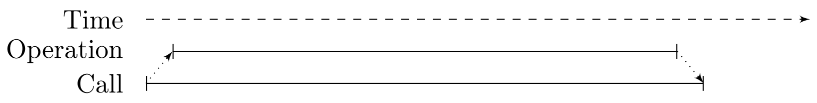
3.3 Semantic terms

The following semantic terms are used throughout the document:

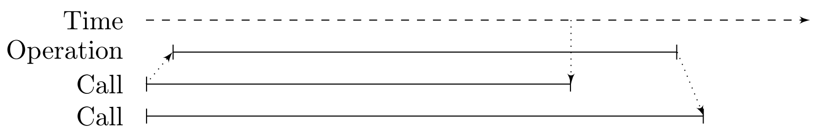
nonblocking A procedure is nonblocking if the procedure may return before the operation completes.



blocking A procedure is blocking if the procedure only returns after the operation has completed.



time-based blocking A procedure is time-based blocking if the procedure may return after the operation completes or after a given timeout has been reached. A corresponding return value is used to distinguish between the two cases.



local A procedure is local if completion of the procedure depends only on the local executing GASPI process.

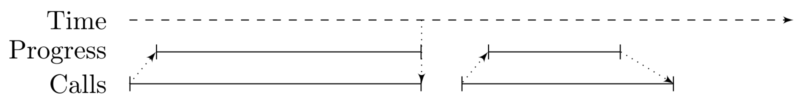
non-local A procedure is non-local if completion of the operation may depend on the existence (and execution) of a remote GASPI process

collective A procedure is collective if all processes in a process group need to invoke the procedure. A collective call may or may not be synchronising.

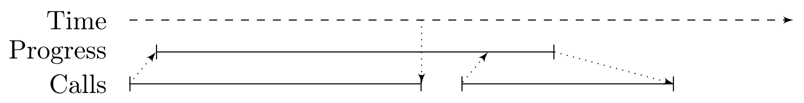
predefined A predefined type is a datatype with a predefined constant name.

timeout A timeout is a mechanism required by procedures that might block (see blocking above). Timeout here is defined as the maximum time (in milliseconds) a called procedure will wait for outstanding communication from other processes. The special value 0 (defined as `GASPI_TEST`) indicates that the procedure will complete all local operations. The procedure subsequently returns the current status without waiting for data from other processes (non-blocking). On the other hand the special value -1 (defined as `GASPI_BLOCK`) instructs the procedure to wait indefinitely (blocking). A number greater than 0 indicates the maximum time the procedure will wait for data from other ranks (time-based blocking). The timeouts hence are soft: The timeout value n does not imply that the called procedure will return after n milliseconds. It just means that the procedure should wait for at most n milliseconds for data from other processes.

synchronous A procedure is called synchronous if progress towards completion only is achieved as long as the application is inside (executing) the procedure.



asynchronous A procedure is called asynchronous if progress towards completion may be achieved after the procedure exits.



Please note that some of the semantic terms are not exclusive. Some of them do overlap. According to the definition, a collective procedure may also be a local procedure. Furthermore, a blocking procedure is per definition also a synchronous procedure; the reverse statement is not true.

3.4 Examples

The examples in this document are for illustration purposes only. They are not intended to specify the semantics.

4 Gaspi concepts

4.1 Introduction and overview

In this section, the basic Gaspi concepts are introduced. A more detailed description with the corresponding procedure specifications can be found in the subsequent topic-specific sections.

Gaspi is a communication API. It implements a Partitioned Global Address Space (*PGAS*) model. Each Gaspi process hosts some part, zero or one or more segments, of the global address space. This partition is local to the process and can be accessed locally just as ordinary allocated memory. In addition to that, every thread on every other Gaspi process can also have access to these segments by read and write operations.

Gaspi was designed with remote direct memory access (RDMA) in mind. A network infrastructure that supports RDMA guarantees asynchronous and one-sided communication operations without involving the CPU. This is one of the main requirements for high scalability which results from interference free communication, e. g. from overlapping communication with computation.

4.2 Gaspi processes

Gaspi provides the concept of ranks. Each Gaspi process receives a unique rank that identifies it during runtime.

4.3 Gaspi groups

Groups are subsets of the set of processes. The group members have common collective operations. A collective operation is then restricted to the processes forming the group.

4.4 Gaspi segments

Modern hardware typically involves a hierarchy of memory with respect to the bandwidth and latencies of read and write accesses. Within that hierarchy are non-uniform memory access (*NUMA*) partitions, solid state devices (*SSD*s), graphical processing unit (*GPU*) memory or many integrated cores (*MIC*) memory. The Gaspi memory segments are supposed to map this variety of hardware layers to the software layer. In the spirit of the PGAS approach, these Gaspi segments may be globally accessible from every thread of every Gaspi process. Gaspi segments can also be used to leverage different memory models within a single application or to even run different applications in a single Partitioned Global Address Space.

4.5 Gaspi one-sided communication

One-sided asynchronous communication is the basic communication mechanism provided by Gaspi. The one-sided communication comes in two flavors. There are read and write operations from and into the Partitioned Global Address Space from arbitrary locations. These operations are non-blocking and asynchronous, allowing the program to continue its execution along the data transfer. The actual data transfer is managed by the underlying network infrastructure.

Gaspi offers the possibility to use different queues to handle the requests. Each communication request can be submitted to one of the supported queues. These queues allow more scalability and can be used as channels for different types of requests where similar types of requests are queued and then get synchronised together but independently from the other ones (separation of concerns). The specification guarantees fairness of transfers posted to different queues, i. e. no queue should see its communication requests delayed indefinitely.

Furthermore, the order of communication request addressing the same remote Gaspi process posted to a given queue are preserved, both on the local as well as on the remote side.

Listing 1: Alltoall with one-sided writes.

```
let nProc be the number of processes;
let iProc be the unique id of this process;
let src[iProc] be an array of size nProc;
let dst[iProc] be an array of size nProc;

foreach process p in [0,nProc):
  write src[iProc][p] into dst[p][iProc];
  //    ^^^^^^^^^^^         ^^^^^^
  //    | local             | remote if p != iProc

wait for the completion of the writes;
// now this process sent the data

barrier;
// now all processes sent the data
```

4.6 GASPI passive communication

Passive communication has a two-sided semantics, where there is a matching receive operation to a send request. Passive communication aims at communication patterns where the sender is unknown (i. e. it can be any process from the receiver perspective) but there is potentially the need for synchronisation between different processes.

The receive operation is a blocking call that has as low interference as possible (e. g. consumes no CPU cycles) and should be woken up directly by the network layer. This passive communication allows for fair distributed updates of globally shared parts of data.

Listing 2: Single consumer and multiple producers using passive communication. The producer transfers a data packet while producing the next data packet, thus overlapping computation and communication.

```
** consumer:
let buffer be one data buffer;
while (!done)
{
  passive_receive into buffer;
  process (buffer);
}

** producer:
let buffer[0] and buffer[1] be two data buffers;
set b to 0;
while (!done)
{
  produce data in buffer[b];

  wait for the completion of earlier passive_send;
  passive_send data from buffer[b] to consumer;

  set b to 1-b;
}
```

4.7 GASPI global atomics

GASPI provides atomic counters, i. e. integral types that can be manipulated through atomic procedures. These atomic procedures are guaranteed to execute from start to end without fear of preemption causing corruption. There are two basic operations on atomic counters: `fetch_and_add` and `compare_and_swap`. The counters can be used as global shared variables and to synchronise processes or events.

Atomic counters are predestined for the implementation of dynamic load balancing schemes for example.

The specification guarantees fairness, i. e. no process should see its atomic operation delayed indefinitely.

Listing 3: Dynamic work distribution: Clients atomically fetch a packet id and increment the value.

```
do
{
  packet := fetch_and_add (1);
  // increment the value by one, return the old value

  if (packet < packet_max): process (packet);
}
while (packet < packet_max);
```

4.8 GASPI collective communication

Collective operations are operations which involve a whole set of GASPI processes. That means that collective operations are collective with respect to a group of processes. They are also exclusive per group, i. e. only one collective operation of a specific type can run at a given time. For example, two allreduce for one group can not run at the same time; however, an allreduce operation and a barrier can run at the same time.

Collective operations can be either synchronous or asnychronous. Synchronous implies that progress is achieved only as long as the application is inside of the call. The call itself, however, may be interrupted by a timeout. The operation is then continued in the next call of the procedure. This implies that a collective operation may involve several procedure calls until completion.

Please note that collective operations can internally also be handled asynchronously, i. e. with progress being achieved outside of the call.

> *Implementor advice:* GASPI does not regulate whether individual collective operations should internally be handled synchronously or asynchronously, however: GASPI aims at an efficient, low-overhead programming model. If asynchronous operation is supported, it should leverage external network-resources, rather than consuming CPU cycles. ⌟

Beside barriers and reductions with predefined operations, reductions with user defined operations are also supported via callback functions.

Collective operations have their own queue and hence typically will be synchronised independently from the operations on other queues (separation of concerns).

4.9 GASPI timeouts

Failure tolerant parallel programs necessitate non-blocking communication calls. Hence, GASPI provides a timeout mechanism for all potentially blocking procedures.

The timeout for a given procedure is specified in milliseconds. `GASPI_BLOCK` is a special predefined timeout value which blocks the invoked procedure until the procedure is completed. This special value shall not be used in a failure tolerant program, because in a situation in which the procedure cannot complete due to a failure on a remote process, the procedure will not return at all.

`GASPI_TEST` is yet another predefined special timeout value which represents a timeout equal to zero. Timeout equal to zero means that the invoked procedure processes an atomic portion of the work and returns after that work has finished. It does not mean that the invoked procedure is doing nothing. It does not mean that the invoked procedure returns immediately.

Example:

```
const gaspi_return_t err = WAIT (..., GASPI_BLOCK);
```

Blocks until the communication queue is empty. No chance to handle failures.

```
const gaspi_return_t err = WAIT (..., GASPI_TEST);
```

Just check if the operation has completed and return as soon as possible.

```
const gaspi_return_t err = WAIT (..., 10);
```

Blocks until the queue is empty or more than 10 milliseconds have passed since the wait call has been initiated.

4.10 GASPI return values

GASPI procedures have three general return values:

```
GASPI_SUCCESS = 0
GASPI_ERROR   = -1
GASPI_TIMEOUT = 1
```

`GASPI_SUCCESS` corresponds to return value equal to zero and implies that the procedure has completed successfully.

`GASPI_TIMEOUT` corresponds to return value equal to one and implies that the procedure could not complete in the given period of time. This does not necessarily mean that an error has occurred. The procedure has to be invoked subsequently in order to fully complete the operation.

`GASPI_ERROR` corresponds to a return value less than zero and implies that the procedure has terminated due to an error. There are no predefined error values specifying the detailed cause of error in the GASPI specification. This is implementation dependent. However, `gaspi_error_message` translates the error code into a human readable format. Moreover, all error codes in the range $(-2, \ldots, -999)$ are reserved for future extensions and must not be used for implementation specific error codes.

> *Implementor advice:* If there are predefined error codes, each of the return codes must have a corresponding error message. ⌟

Additionally, there is a state vector that contains health states for individual processes. The state vector is set after non-local operations and can be used to detect failures on remote processes.

5 GASPI definitions

5.1 Types

typedef unsigned int `gaspi_rank_t`

The GASPI *rank type.* ⌟

typedef unsigned short `gaspi_segment_id_t`

The GASPI *memory segment ID type.* ⌟

typedef unsigned long `gaspi_offset_t`

The GASPI *offset type. Offsets are measured relative to the beginning of a memory segment in units of bytes.* ⌟

typedef unsigned long `gaspi_size_t`

The GASPI *size type. Sizes are measured in units of bytes.* ⌟

typedef unsigned short `gaspi_queue_id_t`

The GASPI *queue ID type.* ⌟

typedef int `gaspi_notification_t`

The GASPI *notification type.* ⌟

typedef unsigned short `gaspi_notification_id_t`

The GASPI *notification ID type.* ⌟

typedef int `gaspi_tag_t`

The GASPI *tag type. Tags are used to discriminate different messages in the passive communication channel.* ⌟

Note! The sum of the sizes of `gaspi_notification_t` and `gaspi_tag_t` should be at most 8 bytes.

typedef unsigned short `gaspi_counter_id_t`

The GASPI *global atomic counter ID type.* ⌟

typedef long `gaspi_counter_value_t`

The GASPI *global atomic counter value type.* ⌟

typedef int `gaspi_return_t`

The GASPI *return value type.* ⌟

typedef `vector<gaspi_return_t> gaspi_returns_t`

The vector type with return codes for individual processes. The length of the vector equals the number of processes in the GASPI *program.* ⌟

typedef int `gaspi_timeout_t`

The GASPI *timeout type.* ⌟

typedef unsigned int `gaspi_number_t`

A type that is used to count elements. That could be numbers of queues as well as the size of individual queues. ⌟

typedef unsigned short `gaspi_group_t`

The GASPI *group type.* ⌟

typedef pointer `gaspi_pointer_t`

A type that can point to some (area of) memory. ⌟

typedef unsigned int `gaspi_alloc_t`

The GASPI *allocation policy type.* ⌟

typedef unsigned int `gaspi_network_t`

The GASPI *network infrastructure type.* ⌟

typedef bool `gaspi_bool_t`

The GASPI *boolean type.* ⌟

typedef char* `gaspi_string_t`

The GASPI *string type.* ⌟

5.2 Constants

```
\enum gaspi_timeout_value_t
\brief special GASPI timeout values

typedef enum { GASPI_BLOCK = -1
             , GASPI_TEST = 0
             } gaspi_timeout_value_t;
```

```
\enum gaspi_return_t
\brief return values

typedef enum { GASPI_SUCCESS = 0
             , GASPI_TIMEOUT = 1
             , GASPI_ERROR   = -1
             } gaspi_return_t;
```

6 Execution model

6.1 Introduction and overview

GASPI does not fix a SPMD (Single Program, Multiple Data) or MPMD (Multiple Program, Multiple Data) style in its approach to parallelism. Hence, either a single program or different programs are started and initialized on the desired target computational units. How the GASPI application is started and initialized is not specified and is implementation specific.

The concept of rank is attributed to each created process. Ranks are a central aspect which allows applications to identify processes and therefore make processing elements act on different tasks or data, even if only a single program is started.

There is also the concept of segments. Segments are memory regions that may be globally available, to be written to or read from. In general, the execution of a GASPI process can be considered as split into several consecutive phases:

- **Setup**

  Setting up configuration parameters (optional)
  Performing environment checks (optional)

- **Initialization**

  Initialization of the runtime environment

- **Working**

  Body of the application
      Communication calls (optional)
      Collective operations (optional)
      Atomic operations (optional)

- **Shutdown**

  Cleanup of communication infrastructure

In the **setup** phase, the application may retrieve and modify the GASPI configuration structure (see section 6.2.1) determining the GASPI runtime behavior. Optionally (but advisable), the application can perform environment checks (see Section 14) to make sure the application can be started safely and correctly.

In the **initialization** phase, the GASPI runtime environment is set up in accordance with the parameters of the configuration structure by invocation of the initialization procedure. The initialization procedure is called before any other functionality, with the exception of pre-initialization routines for environment checking and declaration and retrieval of configuration parameters. After the initialization routine has been called, an optional step to perform is the creation of one or more segments and the creation of one or more groups. Segments are contiguous blocks of memory that may be accessed globally by all processes and where global data should be placed.

After the initialization, the application can proceed with its real **working** phase and use the functionalities of GASPI (communication, collectives, atomic counters, etc.).

Before terminating, the application should call the **shutdown** procedure (see Section 6.3.4) so that the resources and communication infrastructure is cleaned up.

The entire set of execution phases beginning from the setup phase and ending with the shutdown phase defines a GASPI life cycle. In principle, several life cycles can be invoked in one GASPI program.

> *User advice:* Calling a routine in a given execution phase which is not supposed to be executed in that execution phase results in undefined behavior. ⌟

6.2 Process configuration

6.2.1 GASPI configuration structure

The GASPI configuration structure describes the configuration parameters which influence the GASPI runtime behavior.

```
typedef struct {

  // maximum number of groups
  gaspi_number_t     group_max;

  // maximum number of segments
  gaspi_segment_id_t segment_max

  // one-sided comm parameter
  gaspi_number_t     queue_num;
  gaspi_number_t     queue_size_max;
  gaspi_size_t       transfer_size_max;

  // notification parameter
  gaspi_number_t     notification_num;

  // passive comm parameter
  gaspi_number_t     passive_queue_size_max;
  gaspi_size_t       passive_transfer_size_max;

  // atomic counter parameter
  gaspi_number_t     counter_num;

  // collective comm parameter
  gaspi_size_t       allreduce_buf_size;
  gaspi_number_t     allreduce_elem_max;

  // network selection parameter
  gaspi_network_t    network;

  // communication infrastructure build up notification
  gaspi_bool_t       build_infrastructure

  void *             user_defined;

} gaspi_configuration_t;
```

Please note, that for simplicity of notation this is a C-style definition. In bindings to other languages corresponding definitions will be used.

The definition of each of the configuration structure fields is as follows:

group_max the desired maximum number of permissible groups per process.

There is a hardware/implementation dependent maximum.

segment_max the desired number of maximally permissible segments per GASPI process. There is a hardware/implementation dependent maximum.

queue_num the desired number of one-sided communication queues to be created. There is a hardware/implementation dependent maximum.

queue_size_max the desired number of simultaneously allowed on-going requests on a one-sided communication queue. There is a hardware/implementation dependent maximum.

transfer_size_max the desired maximum size of a single data transfer in the one-sided communication channel. There is a hardware/implementation dependent maximum.

notification_num the desired number of internal notification buffers for weak synchronisation to be created. There is a hardware/implementation dependent maximum.

passive_queue_size_max the desired number of simultaneously allowed on-going requests on the passive communication queue. There is a hardware/implementation dependent maximum.

passive_transfer_size_max the desired maximum size of a single data transfer in the passive communication channel. There is a hardware/implementation dependent maximum.

counter_num the desired number of atomic counters to be created. There is a hardware/implementation dependent maximum.

allreduce_elem_max the maximum number of elements in `gaspi_allreduce` There is a hardware/implementation dependent maximum.

allreduce_buf_size the size of the internal buffer of `gaspi_allreduce_user`. There is a hardware/implementation dependent maximum.

network the network type to be used.

build_infrastructure boolean to set whether the communication infrastructure should be built up at startup time. The default value is true.

user_defined some user defined information that is application / implementation dependent.

The default configuration structure can be retrieved by `gaspi_config_get`. Its default values are implementation dependent. If some of the parameters are externally set by the program and committed with `gaspi_config_set`, the requested values are just proposals. Depending on the underlying hardware capabilities, the implementation is allowed to overrule these proposals. `gaspi_config_set` has to be used in order to commit modifications of the configuration structure before the initialization routine is invoked. The actual values of the parameters can be retrieved by the corresponding GASPI getter routines (see section 13) after the successful program initialization. The values of the configuration structure parameters need to be the same on all GASPI processes.

The user has the possibility to set the values on her own or leave the default values. Each field (where applicable) has also a maximum value to avoid user errors that might lead to too much instability or scalability problems (for example, the number of queues). The maximum values applied will be the current GPI values.

6.2.2 `gaspi_config_get`

The `gaspi_config_get` procedure is a *synchronous local blocking* procedure which retrieves the default configuration structure.

```
gaspi_return_t
gaspi_config_get (gaspi_configuration_t config)
```

Parameter:

(out) config: the default configuration

Execution phase:

Setup

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_ERROR`: operation has finished with an error ⌟

After successful procedure completion, i. e. return value `GASPI_SUCCESS`, *config* represents the default configuration.

In case of error, the return value is `GASPI_ERROR`.

6.2.3 `gaspi_config_set`

The `gaspi_config_set` procedure is a *synchronous local blocking* procedure which sets the configuration structure for process initialization.

```
gaspi_return_t
gaspi_config_set (gaspi_configuration_t config)
```

Parameter:

(in) config: the configuration structure to be set

Execution phase:

Setup

Return values:

GASPI_SUCCESS: operation has returned successfully

GASPI_ERROR: operation has finished with an error ⌟

After successful procedure completion, i. e. return value GASPI_SUCCESS, the runtime parameters for the GASPI process initialization are set in accordance with parameters of *config*

In case of error, the return value is GASPI_ERROR.

6.3 Process management calls

6.3.1 gaspi_proc_init

gaspi_proc_init implements the GASPI initialization of the application. It is a *non-local synchronous time-based blocking* procedure.

```
gaspi_return_t
gaspi_proc_init (gaspi_timeout_t timeout)
```

Parameter:

(in) timeout: the timeout.

Execution phase:

Initialization

Return values:

GASPI_SUCCESS: operation has returned successfully

GASPI_TIMEOUT: operation has run into timeout

GASPI_ERROR: operation has finished with an error ⌟

The explicit start of a GASPI process or launch from command line is not specified. This is implementation dependent.

However, it is anticipated that gaspi_proc_init has information about the list of hosts on which the entire GASPI application is running either by environment variables, a command line argument, a daemon or some other mechanism. The actual transfer of knowledge is implementation dependent.

gaspi_proc_init registers a given process at the other remote GASPI processes and sets the corresponding entries in the state vector to a healthy state. If the parameter *build_ infrastructure* in the configuration structure is set, also the communication infrastructure for passive and one-sided communication to all of the other processes is set up. Otherwise, there is no set up of communication infrastructure during the initialization. A rank is assigned to the given GASPI process in accordance with the position of the host in the list. The GASPI process running on the first host in the list has rank zero. The GASPI process running on the second host in the list has rank one and so on.

In case of a node failure, a GASPI process can be started on a new host, freshly allocated or selected from a set of pre-allocated spare hosts, by providing the list of machines in which the failed node is substituted by the new host. The new GASPI process then has the rank of the GASPI process which has been running on the failed node.

In case of the subsequent start of additional GASPI processes, the newly started GASPI process registers with the other remote GASPI processes. Note, that a subsequent change of the number of running GASPI processes invalidates `GASPI_GROUP_ALL` for the old running processes. Also the return value of `gaspi_proc_num` is changed.

The configuration structure should be created by the application before passing it to the `gaspi_proc_init` procedure.

After successful procedure completion, `gaspi_proc_init` returns `GASPI_SUCCESS` and it guarantees that the application has been started on all hosts. In case that the *build_ infrastructure* is set, return value `GASPI_SUCCESS` also implies that the communication infrastructure is up and ready to be used.

In case the application could not be initialized in line with the timeout parameter, the return value is `GASPI_TIMEOUT`. The application is not initialized yet. A subsequent invocation is required to completely initialize the application.

In case of error, the return value is `GASPI_ERROR`. The application is not initialized.

In case of the return value `GASPI_TIMEOUT` or `GASPI_ERROR`, a check of the state vector by invocation of `gaspi_state_vec_get` gives information about whether the involved remote GASPI processes are healthy or whether they are corrupted.

> *Implementor advice:* Calling `gaspi_proc_init` with an enabled parameter *build_ infrastructure* is semantically equivalent to calling `gaspi_proc_init` with a disabled parameter *build_ infrastructure* and subsequent calls to `gaspi_connect` in which an all-to-all connection is established. ⌟

> *User advice:* For resource critical applications, it is recommended to disable the parameter *build_ infrastructure* in the configuration structure. ⌟

> *User advice:* Adjusting the passed list of hosts yields the possibility to take into account the topology of the network. ⌟

> *User advice:* A successful procedure completion does not mean that any communication or collective operation can already be used. Connections might need to be established. A segment has to be allocated for passive communication capabilities. If one-sided communication is supposed to be used, than the segment has to registered in addition. If collective operations are needed, a group has to be created and committed. Only global atomic operations are feasible right from the beginning. ⌟

6.3.2 `gaspi_proc_num`

The total number of GASPI processes started, can be retrieved by `gaspi_proc_num`. This is a *local synchronous blocking* procedure.

```
gaspi_return_t
gaspi_proc_num (gaspi_rank_t proc_num)
```

Parameter:

(out) proc_ num: the total number of GASPI processes

Execution phase:

Working

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_ERROR`: operation has finished with an error ⌟

If successful, the return value is `GASPI_SUCCESS` and `gaspi_proc_num` retrieves the total number of processes that have been initialized and places this number in the *proc_ num*.

In case of error, the return value is `GASPI_ERROR` and the value of *proc_ num* is undefined.

6.3.3 `gaspi_proc_rank`

A rank identifies a GASPI process. The rank of a process lies in the interval $[0, P)$ where P can be retrieved through `gaspi_proc_num`. Each process has a unique rank associated with it. The rank of the invoking GASPI process can be retrieved by `gaspi_proc_rank`. It is a *local synchronous blocking* procedure.

```
gaspi_return_t
gaspi_proc_rank (gaspi_rank_t rank)
```

Parameter:

(out) rank: the rank of the calling GASPI process.

Execution phase:

Working

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_ERROR`: operation has finished with an error ⌟

gaspi_proc_rank retrieves, if successful, the rank of the calling process, placing it in the parameter *rank* and returning GASPI_SUCCESS.

In case of error, the return value is GASPI_ERROR and the value of the *rank* is undefined.

6.3.4 gaspi_proc_term

The shutdown procedure gaspi_proc_term is a *synchronous local time-based blocking* operation that releases resources and performs the required cleanup. There is no definition in the specification of a verification of a healthy global state (i. e. all processes terminated correctly).

After a shutdown call on a given GASPI process, it is undefined behavior if another GASPI process tries to use any non-local GASPI functionality involving that process.

```
gaspi_return_t
gaspi_proc_term (gaspi_timeout_t timeout)
```

Parameter:

(in) timeout: the timeout

Execution phase:

Shutdown

Return values:

GASPI_SUCCESS: operation has returned successfully

GASPI_TIMEOUT: operation has run into timeout

GASPI_ERROR: operation has finished with an error ⌟

In case of successful procedure completion, i. e. return value GASPI_SUCCESS, the allocated GASPI specific resources of the invoking GASPI process have been released. That means in particular that the communication infrastructure is shut down, all committed groups are released and all allocated segments are freed.

In case of timeout, i. e. return value GASPI_TIMEOUT, the local resources of the invoking GASPI process could not be completely released in the given period of time. A subsequent invocation is required to completely release all of the resources.

In case of error, i. e. return value GASPI_ERROR, the resources of the local GASPI process could not be released. The process is in an undefined state.

6.3.5 gaspi_proc_kill

gaspi_proc_kill sends an interrupt signal to a given GASPI process. It is a *synchronous non-local time-based blocking* procedure.

```
gaspi_return_t
gaspi_proc_kill ( gaspi_rank_t rank
                , gaspi_timeout_t timeout
                )
```

Parameter:

(in) rank: the rank of the process to be killed

(in) timeout: the timeout

Execution phase:

Working

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_TIMEOUT`: operation has run into timeout

`GASPI_ERROR`: operation has finished with an error ⌟

`gaspi_proc_kill` sends an interrupt signal to the GASPI process incorporating the rank given by parameter *rank*. This can be used, for example, to realise the registration of a user defined signal handler function which ensures the controlled shut down of an entire GASPI application at the global level if the application receives an interrupt signal (*STRG + C*) in the interactive master process. Every GASPI application should register such or a similar signal handler (c. f. listing 5).

In case of successful procedure completion, i. e. return value `GASPI_SUCCESS`, the remote GASPI process has been terminated.

In case of timeout, i. e. return value `GASPI_TIMEOUT`, the remote GASPI process could not be terminated in the given time. A subsequent invocation of the procedure is needed in order to complete the operation.

In case of error, i. e. return value `GASPI_ERROR`, the state of the remote GASPI process is undefined.

> *User advice:* The kill signal terminates a GASPI process in an uncontrolled way. In order to provide a clean shutdown in this case, it is advisable to be register a user defined signal callback function which guarantees a clean shutdown. ⌟

6.3.6 Example

The listing 4 shows a GASPI "Hello world" example. Please note that this program does not include any code that is needed to deal with failures.

Listing 4: GASPI hello world example.

```
#include <stdio.h>
```

```
#include <stdlib.h>
#include <GASPI.h>

int
main (int argc, char *argv[])
{
  gaspi_proc_init (GASPI_BLOCK);

  gaspi_rank_t iProc = GASPI_NORANK;
  gaspi_rank_t nProc = GASPI_NORANK;

  gaspi_proc_rank (&iProc);
  gaspi_proc_num (&nProc);

  printf ("Hello world from rank %i of %i!\n", iProc, nProc);

  gaspi_proc_term (GASPI_BLOCK);

  return EXIT_SUCCESS;
}
```

The listing 5 shows the registration of a user defined signal handler function which ensures the controlled shut down of an entire GASPI application at the global level if the application receives an interrupt signal ($STRG + C$) in the interactive master process. Every GASPI application should register such or a similar signal handler.

Listing 5: Signal handling.

```
#include <signal.h>
#include <stdlib.h>
#include <GASPI.h>

void
signalHandler (int sigint)
{
  gaspi_rank_t iProc = GASPI_NORANK;
  gaspi_rank_t nProc = GASPI_NORANK;

  gaspi_proc_rank (&iProc);
  gaspi_proc_num (&nProc);

  if (0 == iProc)
    {
      for (iProc = 1; iProc < nProc; ++iProc)
        {
          gaspi_proc_kill (iProc, GASPI_BLOCK);
        }
    }

  gaspi_proc_term (GASPI_BLOCK);

  exit (EXIT_FAILURE);
}


int
main (int argc, char *argv[])
{
  gaspi_proc_init (GASPI_BLOCK);

  signal (SIGINT, &signalHandler);

  /* working phase */

  gaspi_proc_term (GASPI_BLOCK);

  return EXIT_SUCCESS;
}
```

6.4 Connection management utilities

6.4.1 `gaspi_connect`

In order be able to communicate between two Gaspi processes, the communication infrastructure has to be established. This is achieved with the *synchronous non-local time-based blocking* procedure `gaspi_connect`. It is bound to the working phase of the Gaspi life cycle.

```
gaspi_return_t
gaspi_connect ( gaspi_rank_t rank
              , gaspi_timeout_t timeout
              )
```

Parameter:

(in) rank: the remote rank with which the communication infrastructure is established

(in) timeout: The timeout for the operation

Execution phase:

Working

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_TIMEOUT`: operation has run into timeout

`GASPI_ERROR`: operation has finished with an error ⌟

`gaspi_connect` builds up the communication infrastructure, passive as well as one-sided, between the local and the remote GASPI process representing rank *rank*. The connection is bi-directional, i. e. it is sufficient if `gaspi_connect` is invoked by only one of the connection partners.

In case of successful procedure completion, i. e. return value `GASPI_SUCCESS`, the communication infrastructure is established. If there is an allocated segment, the segment can be used as a destination for passive communication between the two nodes. In case the connection has been established already, e. g. by the connection partner, the return value is `GASPI_SUCCESS`.

In case of return value `GASPI_TIMEOUT`, the communication infrastructure could not be established between the local GASPI process and the remote GASPI process in the given period of time.

In case of return value `GASPI_ERROR`, the communication infrastructure could not be established between the local GASPI process and the remote GASPI process.

In case of the latter two return values, a check of the state vector by invocation of `gaspi_state_vec_get` gives information whether the remote GASPI process is still healthy.

> *User advice:* Under the assumption that the GASPI process is initialized with parameter *build_ infrastructure* set to *true*, all the connections are set up at initialization time. Hence, a subsequent call to `gaspi_connect` is superfluous in this case. ⌟

6.4.2 `gaspi_disconnect`

The `gaspi_disconnect` procedure is a *synchronous local blocking* procedure which disconnects a given process, identified by its rank, and frees all associated ressources.

It is bound to the working phase of the GASPI life cycle.

```
gaspi_return_t
gaspi_disconnect ( gaspi_rank_t rank
                 , gaspi_timeout_t timeout
                 )
```

Parameter:

(in) rank: the remote rank from which the communication infrastructure is disconnected

(in) timeout: The timeout for the operation

Execution phase:

Working

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_TIMEOUT`: operation has run into timeout

`GASPI_ERROR`: operation has finished with an error ⌟

`gaspi_disconnect` disconnects the communication infrastructure, passive as well as one-sided, between the local and the remote GASPI process representing rank *rank*. The connection is bi-directional, i. e. it is sufficient if `gaspi_disconnect` is invoked by only one of the connection partners.

In case of successful procedure completion, i. e. return value `GASPI_SUCCESS`, the communication infrastructure is disconnected. Associated resources are freed on the local as well as on the remote side. In case the connection has been disconnected already, e. g. by the connection partner, the return value is `GASPI_SUCCESS`.

In case of error the return value is `GASPI_ERROR`.

In case of return value `GASPI_TIMEOUT`, the connection between the local GASPI process and the remote GASPI process could not be disconnected in the given period of time.

In case of the latter two return values local resources are freed and a check of the state vector by invocation of `gaspi_state_vec_get` gives information whether the remote GASPI process is still healthy.

After successful procedure completion, i. e. return value `GASPI_SUCCESS`, the connection is disconnected and can no longer be used.

6.5 State vector for individual processes

6.5.1 Introduction

A necessary pre-condition for realising a failure tolerant code is a detailed knowledge about the state of the communication partners of each local GASPI process.

GASPI provides a predefined type to describe the state of a remote GASPI process, which is the `gaspi_state_t` type

```
typedef enum { GASPI_STATE_HEALTHY = 0
             , GASPI_STATE_CORRUPT = 1
             } gaspi_state_t;
```

GASPI_STATE_HEALTHY implies that the remote GASPI process is healthy, i. e. communication is possible. GASPI_STATE_CORRUPT means that the remote GASPI process is corrupted, i. e. there is no communication possible.

typedef `vector<gaspi_state_t> gaspi_state_vector_t`

The vector with state information for individual processes. The length of the vector equals the number of processes in the GASPI *program.* ⌟

There are procedures to query the state of the communication partners after a given communication request and also to reset the state after successful recovery. These are described in the following subsections.

6.5.2 `gaspi_state_vec_get`

The state vector is obtained by the *local synchronous blocking* function `gaspi_state_vec_get`.

The state vector represents the states of all GASPI processes.

```
gaspi_return_t
gaspi_state_vec_get (gaspi_state_vector_t state_vector)
```

Parameter:

(out) returns: the vector with individual return codes

Execution phase:

Working

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_ERROR`: operation has finished with an error ⌟

The state vector has one entry for each rank. It is initialized during `gaspi_proc_init`. It is updated after each of the following operations

- process management
  - `gaspi_proc_init`
- group commitment
  - `gaspi_group_commit`
- segment registration
  - `gaspi_segment_register`
- one-sided communication
  - `gaspi_wait`
- passive communication
  - `gaspi_passive_wait`
- collective operations
  - `gaspi_barrier`

  - `gaspi_allreduce`
  - `gaspi_allreduce_user`
- global atomic counters
  - `gaspi_counter_set`
  - `gaspi_counter_fetch_and_add`
  - `gaspi_counter_compare_swap`

`gaspi_state_vec_get` retrieves in case of successful completion, i. e. return value `GASPI_SUCCESS`, the state vector. It contains the states of the GASPI processes with which the local process has been communicating. All other entries are unmodified.

The internal operation to update the state vector is a logic "or".

In case of error, the return value is `GASPI_ERROR` and the value of the state vector is undefined.

> *User advice:* For failure tolerant code, the state vector should be checked after each of the above procedure calls in case they return with either return value `GASPI_ERROR` or `GASPI_TIMEOUT`. ⌟

6.6 MPI Interoperability

GASPI aims at providing interoperability with MPI in order to allow for incremental porting of such applications.

The Start-up of mixed MPI and GASPI code is achieved by invoking `gaspi_proc_init` in an existing MPI program. This way, MPI takes care of distributing and starting the binary and GASPI just takes care of setting up its internal infrastructure.

GASPI and MPI communication should not occur at the same time, i. e. only the program layout given in listing 6 is supported

Listing 6: Embedded GASPI program

```
mpi_startup;

/* MPI part, no ongoing GASPI communication... */

/* ...finish all ongoing MPI communication */

mpi_barrier;

/* no ongoing MPI communication */

gaspi_proc_init;

while (!done) {

  /* GASPI part, no ongoing MPI communication... */

  /* ...finish all ongoing GASPI communication */

  gaspi_barrier;

  /* MPI part, no ongoing GASPI communication... */

  /* ...finish all ongoing MPI communication */

  mpi_barrier;
}

gaspi_proc_term;

/* MPI part, no ongoing GASPI communication */

mpi_shutdown;
```

6.7 Argument checks and performance

GASPI aims at high performance and does not provide any argument checks at procedure invocation per default.

> *Implementor advice:* The implementation should provide a specific library which includes argument checks. The GASPI procedures should include out of bounds checks, there. ⌟

7 Groups

7.1 Introduction

Groups are subsets of the total number of GASPI processes. The group members have common collective operations. Each GASPI process may participate in more than one group.

The use-cases are the collective operations provided in section 12 that make sense to be performed only for a subset of GASPI processes in order to avoid a complete (all processes) collective synchronisation point.

A group has to be defined and declared in each of the participating GASPI processes. Defining a group is a two step procedure. An empty group has to be created first. Then the participating GASPI processes, represented by their ranks, have to be attached. The group definition is a local operation. In order to activate the group, the group has to be committed by each of the participating GASPI processes. This is a collective operation for the group. Only after successful group commitment the group can be used for collective operations.

The maximum number of groups allowed per GASPI process is restricted by the implementation. A desired value can be passed to `gaspi_proc_init` by the configuration structure.

In case one group disappears due to some failure, the group has to be reestablished. If there is a new process replacing the failed one, the group has to be defined and declared on the newly started GASPI process(es). Reestablishment of the group is then achieved by recommitment of the group by the GASPI processes which were still 'alive' (functioning) and by the commitment of the group by the newly started GASPI process.

7.2 GASPI group generics

7.2.1 GASPI group type

Groups are specified with a special group type `gaspi_group_t`.

7.2.2 `GASPI_GROUP_ALL`

`GASPI_GROUP_ALL` is a predefined default group that corresponds to the whole set of GASPI processes. This is to be used for collective operations that work for the whole system.

```
gaspi_group_t GASPI_GROUP_ALL;
```

User advice: Note that `GASPI_GROUP_ALL` is a group definition like any other sub group. In order to be used, `GASPI_GROUP_ALL` also has to be committed by `gaspi_group_commit`. ⌟

7.3 Group creation

7.3.1 `gaspi_group_create`

The `gaspi_group_create` procedure is a *synchronous local blocking* procedure which creates an empty group.

```
gaspi_return_t
gaspi_group_create (gaspi_group_t group)
```

Parameter:

(out) group: the created empty group

Execution phase:

Working

Return values:

GASPI_SUCCESS: operation has returned successfully

GASPI_ERROR: operation has finished with an error ⌟

After successful procedure completion, i. e. return value GASPI_SUCCESS, *group* represents an empty group without any members.

In case of error, the return value is GASPI_ERROR.

7.3.2 gaspi_group_add

The gaspi_group_add procedure is a *synchronous local blocking* procedure which adds a given rank to an existing group.

```
gaspi_return_t
gaspi_group_add ( gaspi_group_t group
                , gaspi_rank_t rank
                )
```

Parameter:

(inout) group: the group to which the rank is added

(in) rank: the rank to add to the group

Execution phase:

Working

Return values:

GASPI_SUCCESS: operation has returned successfully

GASPI_ERROR: operation has finished with an error ⌟

After successful procedure completion, i. e. return value GASPI_SUCCESS, the GASPI process with *rank* is added to *group*.

In case of error, the return value is GASPI_ERROR.

7.3.3 gaspi_group_commit

The gaspi_group_commit procedure is a *synchronous collective time-based blocking* procedure which establishes a group.

```
gaspi_return_t
gaspi_group_commit ( gaspi_group_t group
                   , gaspi_timeout_t timeout
                   )
```

Parameter:

(in) group: the group to commit

(in) timeout: the timeout

Execution phase:

Working

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_TIMEOUT`: operation has run into timeout

`GASPI_ERROR`: operation has finished with an error ⌟

After successful procedure completion, i. e. return value `GASPI_SUCCESS`, the group given by the parameter *group* is established. Collective operations invoked by the members of the group are allowed from this moment on.

In case of timeout, i. e. return value `GASPI_TIMEOUT`, the group could not be established on all ranks forming the group in the given period of time. The group is in an undefined state and collective operations on the group yield undefined behavior. A subsequent invocation is required in order to completely establish the group.

In case of error, i. e. return value `GASPI_ERROR`, the group could not be established. The group is in an undefined state and collective operations defined on the given group yield undefined behavior.

In both cases, `GASPI_TIMEOUT` and `GASPI_ERROR`, the GASPI state vector should be checked in order to eliminate the possibility of a failure.

> *User advice:* Any group commit should be performed only by a single thread of a process. If two GASPI processes are members of two groups, then the order of the group commits should be the same on both processes in order to avoid deadlocks. ⌟

> *Implementor advice:* If the parameter *build_ infrastructure* is not set, the procedure `gaspi_group_commit` must set up the infrastructure for all possible operations of the group. ⌟

7.4 Group deletion

7.4.1 `gaspi_group_delete`

The `gaspi_group_delete` procedure is a *synchronous local blocking* procedure which deletes a given group.

```
gaspi_return_t
gaspi_group_delete (gaspi_group_t group)
```

Parameter:

(in) group: the group to be deleted

Execution phase:

Working

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_ERROR`: operation has finished with an error ⌟

After successful procedure completion, i. e. return value `GASPI_SUCCESS`, *group* is deleted and cannot be used further.

In case of error, the return value is `GASPI_ERROR`.

> *Implementor advice:* If the parameter *build_ infrastructure* is not set to true, the procedure `gaspi_group_delete` must disconnect all connections which have been set up in the call to `gaspi_group_commit` and free all associated resources. ⌟

7.5 Group utilities

7.5.1 `gaspi_group_num`

The `gaspi_group_num` procedure is a *synchronous local blocking* procedure which returns the current number of allocated groups.

```
gaspi_return_t
gaspi_group_num (gaspi_number_t group_num)
```

Parameter:

(out) group_ num: the current number of groups

Execution phase:

Working

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_ERROR`: operation has finished with an error ⌟

After successful procedure completion, i. e. return value `GASPI_SUCCESS`, *group_ num* contains the current number of allocated groups. The value of *group_ num* is related to the parameter *group_ max* in the configuration structure and cannot exceed that value. The value can be implementation specific.

7.5.2 `gaspi_group_size`

The `gaspi_group_size` procedure is a *synchronous local blocking* procedure which returns the number of ranks of a given group.

```
gaspi_return_t
gaspi_group_size ( gaspi_group_t group
                 , gaspi_rank_t group_size
                 )
```

Parameter:

(in) group: the group to be examined

(out) group_ size: the number of ranks in a given group

Execution phase:

Working

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_ERROR`: operation has finished with an error ⌟

After successful procedure completion, i. e. return value `GASPI_SUCCESS`, *group_ size* contains the number of GASPI processes forming the *group.*

In case of error, the return value is `GASPI_ERROR`. The parameter *group_ size* has an undefined value.

7.5.3 `gaspi_group_ranks`

The `gaspi_group_ranks` procedure is a *synchronous local blocking* procedure which returns a list of ranks of GASPI processes forming the group.

```
gaspi_return_t
gaspi_group_ranks ( gaspi_group_t group
                  , gaspi_rank_t group_ranks[group_size]
                  )
```

Parameter:

(in) group: the group to be examined

(out) group_ ranks: the list of ranks forming the group

Execution phase:

Working

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_ERROR`: operation has finished with an error ⌟

After successful procedure completion, i. e. return value `GASPI_SUCCESS`, the list *group_ ranks* contains the ranks of the processes that belong to the *group*. The list is not allocated by the procedure. The list allocation is supposed to be done outside of the procedure. The size of the list can be inquired by `gaspi_group_size`.

In case of error, the return value is `GASPI_ERROR`. The list *group_ ranks* has an undefined value.

8 GASPI segments

8.1 Introduction and overview

Modern hardware has an entire memory hierarchy concerning the bandwidth and latencies of read and write accesses. Among them are non-uniform memory access (*NUMA*) partitions, solid state devices (*SSD*s), graphical processing unit (*GPU*) memory or many integrated cores (*MIC*) memory.

The GASPI memory segments are thus an abstraction representing any kind of memory level, mapping the variety of hardware layers to the software layer. A segment is a contiguous block of virtual memory. In the spirit of the PGAS approach, these GASPI segments may be globally accessible from every thread of every GASPI process and represent the partitions of the global address space.

By means of the GASPI memory segments it is also possible for multiple memory models or indeed multiple applications to share a single Partitioned Global Address Space.

Since segment allocation is expensive and the total number of supported segments is limited due to hardware constraints, the GASPI memory management paradigm is the following. GASPI provides only a few relatively large segments. Allocations inside of the pre-allocated segment memory are managed by the application.

Every GASPI process may possess a certain number of segments (not necessarily equal to the number possessed by the other ranks) that may be accessed as common memory, whether is local—with normal memory operations—or remote—with the communication routines of GASPI.

In order to use a segment for communication between two processes, some setup steps are required in general.

A memory segment has to be allocated in each of the processes by the *local* procedure `gaspi_segment_alloc`. In order to also use the segments for one-sided communication, the memory segment has to be registered on the remote process which will access the memory segment at some point. This is achieved by the *non-local* procedure `gaspi_segment_register`.

> *User advice:* If the parameter *build_ infrastructure* is not set, a connection has to be established between the processes before the segment can be registered at the remote process. This is accomplished by calling the procedure `gaspi_connect`. ⌟

`gaspi_segment_create` unites these steps into a single *collective* procedure for an entire group. After successful procedure completion, a common segment is created on each GASPI process forming the group which can be immediately used for communication among the group members.

During the lifetime of an application there is no segment available unless it is explicitly created with `gaspi_segment_alloc` or `gaspi_segment_create` after the GASPI start-up.

8.2 Segment creation

8.2.1 `gaspi_segment_alloc`

The *synchronous local blocking* procedure `gaspi_segment_alloc` allocates a memory segment and optionally maps it in accordance with a given allocation policy.

```
gaspi_return_t
gaspi_segment_alloc ( gaspi_segment_id_t segment_id
                    , gaspi_size_t size
                    , gaspi_alloc_t alloc_policy
                    )
```

Parameter:

(in) segment_ id: The segment ID to be created. The segment IDs need to be unique on each GASPI process

(in) size: The size of the segment in bytes

(opt) alloc_policy: optional parameter characterizing the allocation policy

Execution phase:

Working

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_ERROR`: operation has finished with an error ⌟

`gaspi_segment_alloc` allocates a segment of size *size* that will be referenced by the *segment_ id* identifier. This identifier parameter has to be unique in the local GASPI process. Creating a new segment with an existing segment ID results in undefined behavior. Note that the total number of segments is restricted by the underlying hardware capabilities. The maximum number of supported segments can be retrieved by invoking `gaspi_segment_max`. The procedure has an optional parameter which might be used to pass an allocation policy. This is left to the implementation.

After successful procedure completion, i. e. return value `GASPI_SUCCESS`, the segment can be accessed locally. In case that there is a connection established to a remote GASPI process, it can also be used for passive communication between the two GASPI processes. (Note that this is always the case if the process has been initialized with the parameter *build_ infrastructure* set to *true*), it can also be used for passive communication between the two GASPI processes; either as a source segment for `gaspi_passive_send` or as a destination segment for `gaspi_passive_receive`.

A return value `GASPI_ERROR` indicates that the segment allocation failed. The segment cannot be used locally or for passive communication.

> *Implementor advice:* In case of non-uniform memory access architectures, the memory should be allocated close to the calling process. The allocation policy of the calling process should not be modified. ⌟

8.2.2 `gaspi_segment_register`

In order to be used in a one-sided communication request on an existing connection, a segment allocated by `gaspi_segment_alloc` needs to be made visible and accessible for the other GASPI processes. This is accomplished by the procedure `gaspi_segment_register`. It is a *synchronous non-local time-based blocking* procedure.

```
gaspi_return_t
gaspi_segment_register ( gaspi_segment_id_t segment_id
                       , gaspi_rank_t rank
                       , gaspi_timeout_t timeout
                       )
```

Parameter:

(in) segment_ id: The segment ID to be registered. The segment ID's need to be unique for each GASPI process

(in) rank: The rank of the GASPI process which should register the new segment

(in) timeout: The timeout for the operation

Execution phase:

Working

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_TIMEOUT`: operation has run into timeout

`GASPI_ERROR`: operation has finished with an error ⌟

`gaspi_segment_register` makes the segment referenced by the *segment_id* identifier visible and accessible to the GASPI process with the associated *rank*.

> *User advice:* If the parameter *build_infrastructure* is not set, a connection has to be established between the processes before the segment can be registered at the remote process. This is accomplished calling the procedure `gaspi_connect`. ⌟

In case of successful procedure completion, i. e. return value `GASPI_SUCCESS`, the local segment can be used for one-sided communication requests which are invoked by the given remote process.

In case of return value `GASPI_TIMEOUT`, the segment could not be registered in the given period of time. The segment cannot be used for one-sided communication requests which are invoked by the given remote process. A subsequent call of `gaspi_segment_register` has to be invoked in order to complete the registration request.

In case of return value `GASPI_ERROR`, the segment could not be registered on the remote side. The segment cannot be used for one-sided communication requests which are invoked by the given remote process.

In case of the latter two return values, a check of the state vector by invocation of `gaspi_state_vec_get` gives information as to whether or not the remote GASPI process is still healthy.

> *User advice:* Note that a local return value `GASPI_SUCCESS` does not imply that the remote process is informed explicitly that the segment is accessible. This can be achieved through an explicit synchronisation, either by one of the collective operations or by an explicit notification. ⌟

8.2.3 `gaspi_segment_create`

`gaspi_segment_create` is a *synchronous collective time-based blocking* procedure. It is semantically equivalent to a collective aggregation of `gaspi_segment_alloc`, `gaspi_segment_register` and `gaspi_barrier` involving all of the members of a given group. If the communication infrastructure was not established for all group members beforehand, `gaspi_segment_create` will accomplish this as well.

```
gaspi_return_t
gaspi_segment_create ( gaspi_segment_id_t segment_id
                     , gaspi_size_t size
                     , gaspi_group_t group
                     , gaspi_timeout_t timeout
                     , gaspi_alloc_t alloc_policy
                     )
```

Parameter:

(in) segment_id: The ID for the segment to be created. The segment ID's need to be unique for each GASPI process

(in) size: The size of the segment in bytes

(in) group: The group which should create the segment

(in) timeout: The timeout for the operation

(opt) alloc_policy: optional parameter characterizing the allocation policy

Execution phase:

Working

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_TIMEOUT`: operation has run into timeout

`GASPI_ERROR`: operation has finished with an error ⌟

`gaspi_segment_create` allocates a segment of size *size* that will be referenced by the *segment_id* identifier. This identifier parameter has to be unique on the local GASPI process. Creating a new segment with an existing segment ID results in undefined behavior. `gaspi_segment_create` makes the segment referenced by the *segment_id* identifier visible and accessible to all of the GASPI processes forming the group *group*. The maximum number of supported segments can be retrieved by invoking `gaspi_segment_max`. The procedure has an optional parameter which might be used to pass an allocation policy. This is left to the implementation.

After successful procedure completion, i. e. `GASPI_SUCCESS`, the segment can be accessed locally and it can be used as a destination for the passive communication channel. Either as a source segment for `gaspi_passive_send` or as a destination segment for `gaspi_passive_receive`. Furthermore, it can be used for one-sided communication requests, which are invoked by the remote processes forming the group *group*. The segment *segment_id* is ready to be used.

For consistency and programs with hard failure tolerance requirements, the operation must be performed within *timeout* milliseconds. In case of return value `GASPI_TIMEOUT`, progress has been achieved, however the operation could not be completed in the given timeout. The segment cannot be used locally or for passive communication. Furthermore, the segment cannot be used for one-sided communication requests which are invoked by the other remote processes forming the group. A subsequent call of `gaspi_segment_create` has to be invoked in order to complete the segment creation.

In case of return value `GASPI_ERROR`, the segment creation failed in one of the above progress steps on at least one of the involved GASPI processes. The segment cannot be used locally or for passive communication. Furthermore, the segment cannot be used for one-sided communication requests which are invoked by the other remote processes forming the group.

In case of the latter two return values, a check of the state vector by invocation of `gaspi_state_vec_get` gives information whether the involved remote GASPI processes are still healthy.

> *Implementor advice:* In case of non-uniform memory access architectures, the memory should be allocated close to the calling process. The allocation policy of the calling process should not be modified. ⌟

8.3 Segment deletion

8.3.1 `gaspi_segment_delete`

The *synchronous local blocking* procedure `gaspi_segment_delete` releases the resources of a previously allocated memory segment.

```
gaspi_return_t
gaspi_segment_delete (gaspi_segment_id_t segment_id)
```

Parameter:

(in) segment_ id: The segment ID to be deleted.

Execution phase:

Working

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_ERROR`: operation has finished with an error ⌟

`gaspi_segment_delete` releases the resources of the segment which is referenced by the *segment_ id* identifier.

After successful procedure completion, i. e. return value `GASPI_SUCCESS`, the segment is deleted and the resources are released. It would be an application error to use the segment for communication between two GASPI processes after `gaspi_delete` has been called.

In case of return value `GASPI_ERROR`, the segment deletion failed. The segment is in an undefined state. It cannot be used locally or for passive communication. Furthermore, the segment cannot be used for one-sided communication requests which are invoked by other remote processes.

8.4 Segment utilities

8.4.1 `gaspi_segment_num`

The `gaspi_segment_num` procedure is a *synchronous local blocking* procedure which returns the current number of allocated segments.

```
gaspi_return_t
gaspi_segment_num (gaspi_segment_id_t segment_num)
```

Parameter:

(out) segment_num: the current number of allocated segments

Execution phase:

Working

Return values:

GASPI_SUCCESS: operation has returned successfully

GASPI_ERROR: operation has finished with an error ⌟

After successful procedure completion, i. e. return value GASPI_SUCCESS, *segment_num* contains the current number of allocated segments provided by Gaspi. The value of *segment_num* is related to the parameter *segment_max* in the configuration structure which is retrieved by gaspi_proc_init at startup and cannot exceed that value. The maximum number of allocatable segments per process might be implementation specific.

In case of error, the return value is GASPI_ERROR. The parameter *segment_num* has an undefined value.

8.4.2 gaspi_segment_ptr

Segments are identified by a unique ID. This ID can be used to obtain the virtual address of that local segment of memory. The procedure gaspi_segment_ptr returns the pointer to the segment represented by a given segment ID. It is a *synchronous local blocking* procedure.

```
gaspi_return_t
gaspi_segment_ptr ( gaspi_segment_id_t segment_id
                  , gaspi_pointer_t pointer
                  )
```

Parameter:

(in) segment_id: The segment ID.

(out) pointer: The pointer to the memory segment.

Execution phase:

Working

Return values:

GASPI_SUCCESS: operation has returned successfully

`GASPI_ERROR`: operation has finished with an error ⌟

After successful procedure completion, i. e. `GASPI_SUCCESS`, the output parameter *pointer* contains the virtual address pointer of the memory identified by *segment_ id*. This `gaspi_pointer_t` can then be used to reference the segment and perform memory operations.

In case of return value `GASPI_ERROR`, the translation of the segment ID to a pointer to a virtual memory address failed. The pointer contains an undefined value and cannot be used to reference the segment.

8.5 Segment memory management

Each thread of a process may have global read or write access to all of the segments provided by remote GASPI processes if there is a connection established between the processes and if the respective segments have been registered on the local process.

Since a segment is an entire contiguous block of virtual memory, allocations inside of the pre-allocated segment memory need to be managed.

GASPI does not provide dedicated memory management functionality for the local segments. This is left to the application. High performance is the guiding principle of GASPI. A good problem-related implementation of a memory management is always better than any predefined implementation. A default implementation cannot include knowledge about the specific problem.

Local and *non-local* GASPI procedures specify in general memory addresses within the Partitioned Global Address Space by the triple consisting of a rank, a segment identifier and an offset. This prevents a global all-to-all distribution of memory addresses, since memory addresses of memory segments could be and normally are different on different GASPI processes.

A local buffer is specified by the pair *segment_ id*, *offset*. The rank parameter is superfluous since it can be derived from the rank of the local GASPI process. The buffer is located at address

$$\text{buffer\_address} = \text{base\_addr}\,(\,\text{segment\_id}\,) + \text{offset}$$

where base_addr(segment_id) is the base address of the segment with identifier *segment_ id*. It can be obtained by applying `gaspi_segment_ptr` on the local process.

A remote buffer is specified by the triple *remote_ rank*, *remote_ segment_ id*, *remote_ offset*. The address of the remote buffer can be calculated analogously to the local buffer. The only difference is the determination of the base address. Here, it is the address which would be obtained by invoking `gaspi_segment_ptr` on the remote GASPI process with *remote_ segment_ id* as input parameter.

9 One-sided communication

9.1 Introduction and overview

One-sided asynchronous communication is the basic communication mechanism provided by GASPI. Hereby, one GASPI process specifies all communication parameters, both for the local and the remote side. Due to the asynchronicity, a complete communication involves two procedure calls. First, one call to initiate the communication. This call posts a communication request to the underlying network infrastructure. The second call waits for the completion of the communication request.

For one-sided communication, GASPI provides the concept of communication queues. All operations placed on a certain queue q by one or several threads are finished after a single wait call on the queue q has returned successfully. Separation of concerns is possible by using different queues for different tasks, e. g. one queue for operations on data and another queue for operations on meta-data.

The different communication queues guarantee fair communication, i. e. no queue should see its communication requests delayed indefinitely. Furthermore, a single queue preserves the order of the communication requests on the local and the remote side if the remote rank of two requests is the same.

One-sided communication calls can basically be divided into two operation types: read and write. The read operations transfer data from a remote segment to a local segment. For the write operation, it is vice versa.

The number of communication queues and their size can be configured at initialization time, otherwise default values will be used. The default values are implementation dependent. Maximum values are also defined.

Due to the direct memory access of one-sided communication calls, some notes regarding the memory addresses should be made: Memory addresses within the Partitioned Global Address Space are specified uniquely by the triple consisting of the rank, segment identifier and the offset. This prevents a global all-to-all distribution of memory addresses, since the memory addresses of memory segments could be and normally are different on different GASPI processes.

For the write operation there are four different variants that allow different communication patterns:

- `gaspi_write`
- `gaspi_write_notify`
- `gaspi_write_list`
- `gaspi_write_list_notify`

The read operations have two different variants that allow different communication patterns:

- `gaspi_read`
- `gaspi_read_list`

The read operations do not support notification calls. This is due to the fact that a notification can only be transferred after ensuring that the communication request has been processed. This would imply that a subsequent wait call has to be invoked directly after invoking read. However, this can be managed more effectively by the application.

A valid one-sided communication request requires that the local and the remote segment are allocated, that there is a connection between the local and the remote GASPI process and that the remote segment has been registered on the local GASPI process.

9.2 Basic communication calls

9.2.1 `gaspi_write`

The simplest form of a write operation is `gaspi_write` which is a single communication call to write data to a remote location. It is an *asynchronous non-local time-based blocking* procedure.

```
gaspi_return_t
gaspi_write ( gaspi_segment_id_t segment_id_local
            , gaspi_offset_t offset_local
            , gaspi_rank_t rank
            , gaspi_segment_id_t segment_id_remote
            , gaspi_offset_t offset_remote
            , gaspi_size_t size
            , gaspi_queue_id_t queue
            , gaspi_timeout_t timeout
            )
```

Parameter:

(in) segment_id_local: the local segment ID to read from

(in) offset_local: the local offset in bytes to read from

(in) rank: the remote rank to write to

(in) segment_id_remote: the remote segment to write to

(in) offset_remote: the remote offset to write to

(in) size: the size of the data to write

(in) queue: the queue to use

(in) timeout: the timeout

Execution phase:

Working

Return values:

GASPI_SUCCESS: operation has returned successfully

GASPI_TIMEOUT: operation has run into timeout

GASPI_ERROR: operation has finished with an error ⌟

gaspi_write posts a communication request which asynchronously transfers a contiguous block of *size* bytes from a source location of the local GASPI process to a target location of a remote GASPI process. This communication request is posted to the communication queue *queue*. The source location is specified by the pair *segment_id_local*, *offset_local*. The target location is specified by the triple *rank*, *segment_id_remote*, *offset_remote*.

A valid gaspi_write communication request requires that the local and the remote segment are allocated, that there is a connection between the local and the remote GASPI process and that the remote segment has been registered on the local GASPI process. Otherwise, the communication request is invalid and the procedure returns with GASPI_ERROR.

After successful procedure completion, i. e. return value GASPI_SUCCESS, the communication request has been posted to the underlying network infrastructure. One new entry is inserted into the given queue.

Successive gaspi_write calls posted to the same queue and the same destination rank are non-overtaking. Non-overtaking means that the order of communication requests is preserved on the remote side. In particular, one can assume, that if the data from the later request has arrived on the remote process, also the data from the earlier posted request to the same process have arrived on the remote side.

gaspi_write calls may be posted from every thread of the GASPI process.

If the procedure returns with GASPI_TIMEOUT, the communication request could not be posted to the hardware during the given timeout. This can happen, if another thread is in a gaspi_wait for the same queue. A subsequent call of gaspi_write has to be invoked in order to complete the write call.

A communication request posted to a given queue can be considered as completed, if the correspondent gaspi_wait returns with GASPI_SUCCESS.

If the queue to which the communication request is posted is full, i. e. if the number of posted communication requests has already reached the queue size of a given queue, the communication request fails and the procedure returns with return value GASPI_ERROR. If a saturated queue is detected, there are the following two options: Either one invokes a gaspi_wait on the given queue in order to wait for all the posted requests to be finished. Or one tries to use another queue.

User advice: Return value `GASPI_SUCCESS` does not mean, that the data has been transferred or buffered or that the data has arrived at the remote side.
It is allowed to write data to the source location while the communication is ongoing. However, the result on the remote side would be some undefined interleaving of the data that was present when the call was issued and the data that was written later.
It is also allowed to read from the source location while the communcation is ongoing and such a read would retrieve the data written by the application.
Use `gaspi_notify` to synchronise the communication. ⌟

9.2.2 `gaspi_read`

The simplest form of a read operation is `gaspi_read` which is a single communication call to read data from a remote location. It is an *asynchronous non-local time-based blocking* procedure.

```
gaspi_return_t
gaspi_read ( gaspi_segment_id_t segment_id_local
           , gaspi_offset_t offset_local
           , gaspi_rank_t rank
           , gaspi_segment_id_t segment_id_remote
           , gaspi_offset_t offset_remote
           , gaspi_size_t size
           , gaspi_queue_id_t queue
           , gaspi_timeout_t timeout
           )
```

Parameter:

(in) segment_id_local: the local segment ID to write to

(in) offset_local: the local offset in bytes to write to

(in) rank: the remote rank to read from

(in) segment_id_remote: the remote segment to read from

(in) offset_remote: the remote offset to read from

(in) size: the size of the data to read

(in) queue: the queue to use

(in) timeout: the timeout

Execution phase:

Working

Return values:

`GASPI_SUCCESS`: operation has returned successfully

GASPI_TIMEOUT: operation has run into timeout

GASPI_ERROR: operation has finished with an error ⌟

gaspi_read posts a communication request which asynchronously transfers a contiguous block of *size* bytes from a source location of a remote GASPI process to a target location of the local GASPI process. This communication request is posted to the communication queue *queue*. The target location is specified by the pair *segment_id_local*, *offset_local*. The source location is specified by the triple *rank*, *segment_id_remote*, *offset_remote*.

A valid gaspi_read communication request requires that the local and the remote segment are allocated, that there is a connection between the local and the remote GASPI process and that the remote segment has been registered on the local GASPI process. Otherwise, the communication request is invalid and the procedure returns with GASPI_ERROR.

After successful procedure completion, i. e. return value GASPI_SUCCESS, the communication request has been posted to the underlying network infrastructure. One new entry is inserted into the given queue.

Successive gaspi_read calls posted to the same queue and the same destination rank are non-overtaking. Non-overtaking means that the order of communication requests is preserved. In particular, one can assume, that if the data from the later request has arrived, also the data from the earlier posted request to the same process have arrived.

gaspi_read calls may be posted from every thread of the GASPI process.

If the procedure returns with GASPI_TIMEOUT, the communication request could not be posted to the hardware during the given timeout. This can happen, if another thread is in a gaspi_wait for the same queue. A subsequent call of gaspi_read has to be invoked in order to complete the read call.

A communication request posted to a given queue can be considered as completed, if the the correspondent gaspi_wait returns with GASPI_SUCCESS.

If the queue to which the communication request is posted is full, i. e. that the number of posted communication requests has already reached the queue size of a given queue, the communication request fails and the procedure returns with return value GASPI_ERROR. If a saturated queue is detected, there are the following two options: Either one invokes a gaspi_wait on the given queue in order to wait for all the posted requests to be finished. Or one tries to use another queue.

User advice: Return value `GASPI_SUCCESS` does not mean, that the data transfer has started or that the data has been received at the local side. It is allowed to write data to the local target location while the communication is ongoing. However, the content of the memory would be some undefined interleaving of the data transferred from remote side and the data written locally.
Also, it is allowed to read from the local target location while the communication is ongoing. Such a read would retrieve some undefined interleaving of the data that was present when the call was issued and the data that was transferred from the remote side.
Use `gaspi_notify` to synchronise the communication. ⌟

9.2.3 gaspi_wait

The `gaspi_wait` procedure is a time-based blocking local procedure which waits until all one-sided communication requests posted to a given queue are processed by the network infrastructure. It is an *asynchronous non-local time-based blocking* procedure.

```
gaspi_return_t
gaspi_wait ( gaspi_queue_id_t queue
           , gaspi_timeout_t timeout
           )
```

Parameter:

(in) queue: the queue ID to wait for

(in) timeout: the timeout

Execution phase:

Working

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_TIMEOUT`: operation has run into timeout

`GASPI_ERROR`: operation has finished with an error ⌟

After successful procedure completion, i. e. return value `GASPI_SUCCESS`, the hitherto posted communication requests have been processed by the network infrastructure and the queue is cleaned up. After that, any communication request which has been posted to the given queue can be considered as completed on the local side.

`gaspi_wait` procedure calls may be posted from every thread of the local GASPI process. However, the wait operation is a thread exclusive operation and therefore needs privileged access to the queue which means that if a write/read is done while a wait is in operation, the write/read operation blocks to ensure correctness. Enforcing this provides correctness and safety to the user while being easier for the implementor and still allows for a high performance implementation. As a consequence, successive `gaspi_wait` calls invoked for the same queue by different threads are processed in some sequence one after another.

If the procedure returns with `GASPI_TIMEOUT`, the wait request could not be completed during the given timeout. This can happen, if there is another thread in a `gaspi_wait` for the same queue. A subsequent call of `gaspi_wait` has to be invoked in order to complete the call.

If the procedure returns with `GASPI_ERROR`, the wait request aborted abnormally.

In both cases, `GASPI_TIMEOUT` and `GASPI_ERROR`, the GASPI state vector should be checked in order to eliminate the possibility of a failure. If a failure is detected, all of the communication requests which have been posted to the given queue since the last `gaspi_wait` are in an undefined state. Here, undefined state means that the local GASPI process does not know which requests have been processed and which requests are still outstanding. A call to `gaspi_queue_purge` has to be invoked in order to reset the queue.

> *User advice:* Return value `GASPI_SUCCESS` means, that the data of all posted write requests has been transferred to the remote side. It does not mean, that the data has arrived at the remote side. However, write accesses to the local source location will not affect the data that is placed in the remote target location. ⌟

> *User advice:* Return value `GASPI_SUCCESS` means, that the data of all posted read requests have arrived at the local side. ⌟

9.2.4 Examples

Listing 7 shows a matrix transpose of a distributed square matrix implemented with the function `gaspi_write`.

Listing 7: GASPI all-to-all communication (matrix transpose) implemented with `gaspi_write`

```
#include <stdlib.h>
#include <GASPI.h>
#include <success_or_die.h>
#include <wait_if_queue_full.h>

extern void dump (int *arr, int nProc);

int
main (int argc, char *argv[])
{
  ASSERT (gaspi_proc_init (GASPI_BLOCK));

  gaspi_rank_t iProc = GASPI_NORANK;
  gaspi_rank_t nProc = GASPI_NORANK;

  ASSERT (gaspi_proc_rank (&iProc));
  ASSERT (gaspi_proc_num (&nProc));

  gaspi_notification_id_t notification_max;
  ASSERT (gaspi_notification_num(&notification_max));

  if (notification_max < (gaspi_notification_id_t)nProc)
    {
      exit (EXIT_FAILURE);
    }

  ASSERT (gaspi_group_commit (GASPI_GROUP_ALL, GASPI_BLOCK));

  const gaspi_segment_id_t segment_id_src = 0;
  const gaspi_segment_id_t segment_id_dst = 1;

  const gaspi_size_t segment_size = nProc * sizeof(int);

  ASSERT (gaspi_segment_create ( segment_id_src, segment_size
                               , GASPI_GROUP_ALL, GASPI_BLOCK
                               )
         );
  ASSERT (gaspi_segment_create ( segment_id_dst, segment_size
                               , GASPI_GROUP_ALL, GASPI_BLOCK
                               )
         );

  int *src = NULL;
  int *dst = NULL;

  ASSERT (gaspi_segment_ptr (segment_id_src, &src));
  ASSERT (gaspi_segment_ptr (segment_id_dst, &dst));

  const gaspi_queue_id_t queue_id = 0;

  for (gaspi_rank_t rank = 0; rank < nProc; ++rank)
    {
      src[rank] = iProc * nProc + rank;

      const gaspi_offset_t offset_src = rank * sizeof (int);
      const gaspi_offset_t offset_dst = iProc * sizeof (int);
      const gaspi_notification_id_t notify_ID = rank;

      wait_if_queue_full (queue_id, 2);

      const gaspi_notification_t notify_val = 1;

      ASSERT
        (gaspi_write_notify ( segment_id_src, offset_src
                             , rank, segment_id_dst, offset_dst
                             , sizeof (int), notify_ID, notify_val
                             , queue_id, GASPI_BLOCK
                             )
        );
    }

  gaspi_notification_id_t notify_cnt = nProc;
  const gaspi_notification_id_t notify_ID_max = nProc;

  while (notify_cnt > 0)
  {
    ASSERT (gaspi_notify_waitsome (0, nProc, GASPI_BLOCK));

    for ( gaspi_notification_id_t notify_ID = 0
        ; notify_ID < notify_ID_max
        ; ++notify_ID
        )
      {
        gaspi_notification_id_t notify_val = 0;

        ASSERT (gaspi_notify_reset (notify_ID, &notify_val));

        if (notify_val != 0)
          {
            --notify_cnt;
          }
      }
  }

  dump (dst, nProc);

  ASSERT (gaspi_wait (queue_id, GASPI_BLOCK));

  ASSERT (gaspi_barrier (GASPI_GROUP_ALL, GASPI_BLOCK));

  ASSERT (gaspi_proc_term (GASPI_BLOCK));

  return EXIT_SUCCESS;
}
```

Listing 8 shows a matrix transpose of a distributed square matrix implemented with the function `gaspi_read`. Please note the differences between the transpose implemented with write and the transpose implemented with read: The implementation using write can initialize the matrix on-the-fly, right before the data is transferred, while the implementation using read has to synchronise all processes after the local initialization in order to be sure to read valid data. On the other hand, in the implementation using write one has to synchronise after the local wait whereas in the implementation using read one can directly use the data after the local wait returns.

Listing 8: GASPI all-to-all communication (matrix transpose) implemented with gaspi_read

```
#include <stdlib.h>
#include <GASPI.h>
#include <success_or_die.h>
#include <wait_if_queue_full.h>

extern void dump (int *arr, int nProc);

int
main (int argc, char *argv[])
{
  ASSERT (gaspi_proc_init (GASPI_BLOCK));

  gaspi_rank_t iProc = GASPI_NORANK;
  gaspi_rank_t nProc = GASPI_NORANK;

  ASSERT (gaspi_proc_rank (&iProc));
  ASSERT (gaspi_proc_num (&nProc));

  ASSERT (gaspi_group_commit (GASPI_GROUP_ALL, GASPI_BLOCK));

  const gaspi_segment_id_t segment_id_src = 0;
  const gaspi_segment_id_t segment_id_dst = 1;

  const gaspi_size_t segment_size = nProc * sizeof(int);

  ASSERT (gaspi_segment_create ( segment_id_src, segment_size
                               , GASPI_GROUP_ALL, GASPI_BLOCK
                               )
         );
  ASSERT (gaspi_segment_create ( segment_id_dst, segment_size
                               , GASPI_GROUP_ALL, GASPI_BLOCK
                               )
         );

  int *src = NULL;
  int *dst = NULL;

  ASSERT (gaspi_segment_ptr (segment_id_src, &src));
  ASSERT (gaspi_segment_ptr (segment_id_dst, &dst));

  const gaspi_queue_id_t queue_id = 0;

  for (gaspi_rank_t rank = 0; rank < nProc; ++rank)
    {
      src[rank] = iProc * nProc + rank;
    }

  ASSERT (gaspi_barrier (GASPI_GROUP_ALL, GASPI_BLOCK));

  for (gaspi_rank_t rank = 0; rank < nProc; ++rank)
    {
      const gaspi_offset_t offset_src = iProc * sizeof (int);
      const gaspi_offset_t offset_dst = rank * sizeof (int);

      wait_if_queue_full (queue_id, 1);

      ASSERT (gaspi_read ( segment_id_dst, offset_dst
                         , rank, segment_id_src, offset_src
                         , sizeof (int), queue_id, GASPI_BLOCK
                         )
             );
    }

  ASSERT (gaspi_wait (queue_id, GASPI_BLOCK));

  dump (dst, nProc);

  ASSERT (gaspi_barrier (GASPI_GROUP_ALL, GASPI_BLOCK));

  ASSERT (gaspi_proc_term (GASPI_BLOCK));

  return EXIT_SUCCESS;
}
```

The definition of the macro `ASSERT` is given in the listings 11 and 12 starting on page 100. The definition of the function `wait_if_queue_full` is given in the listings 13 and 14 starting on page 101.

9.3 Weak synchronisation primitives

9.3.1 Introduction

The one-sided communication procedures have the characteristics that the entire communication is managed by the local process only. The remote process is not involved. This has the advantage that there is no inherent synchronisation between the local and the remote process in every communication request. However, at some point, the remote process needs the information as to whether the data which has been sent to that process has arrived and is valid.

Therefore, GASPI provides so-called weak synchronisation primitives which allows the application to inform the remote side that the data has been transferred by updating a notification on the remote side. These notifications must be attached to the same queue to which the data payload has been attached. Otherwise, causality is not guaranteed.

As counterpart, there are routines which wait for an update of a single or even an entire set of notifications. This is not thread save. There is a thread safe atomic function to reset the local notification with a given ID which returns the value of the notification before it is reset.

These notification procedures are also one-sided and involve only the local process.

9.3.2 `gaspi_notify`

`gaspi_notify` is an *asynchronous non-local time-based blocking* procedure.

```
gaspi_return_t
gaspi_notify ( gaspi_rank_t rank
             , gaspi_notification_id_t notification_id
             , gaspi_notification_t notification_value
             , gaspi_queue_id_t queue
             , gaspi_timeout_t timeout
             )
```

Parameter:

(in) rank: the remote rank to notify

(in) notification_ id: the remote notification ID

(in) notification_ value: the notification value (> 0) to write

(in) queue: the queue to use

(in) timeout: the timeout

Execution phase:

Working

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_TIMEOUT`: operation has run into timeout

`GASPI_ERROR`: operation has finished with an error ⌟

`gaspi_notify` posts a notification request which asynchronously transfers the notification *notification_ value* of the local GASPI process to an internal notification buffer of a remote GASPI process. This notification request is posted to the communication queue *queue*. The remote notification buffer is specified by the pair *rank*, *notification_ id*.

A valid `gaspi_notify` communication request requires that there is a connection between the local and the remote GASPI process. Otherwise, the communication request is invalid and the procedure returns with `GASPI_ERROR`.

After successful procedure completion, i. e. return value `GASPI_SUCCESS`, the notification request has been posted to the underlying network infrastructure. One new entry is inserted into the given queue.

Successive gaspi_write and gaspi_notify calls posted to the same queue and the same destination rank are non-overtaking. Non-overtaking means that the order of communication requests is preserved on the remote side. In particular, one can assume, that if the data from the later request has arrived on the remote process, also the data from the earlier posted request to the same process have arrived on the remote side.

gaspi_notify calls may be posted from every thread of the GASPI process.

If the procedure returns with GASPI_TIMEOUT, the notification request could not be posted to the hardware during the given timeout. This can happen if another thread is in a gaspi_wait for the same queue. A subsequent call of gaspi_notify has to be invoked in order to complete the call.

A notification request posted to a given queue can be considered as completed, if the the correspondent gaspi_wait returns with GASPI_SUCCESS.

If the queue to which the communication request is posted is full, i. e. that the number of posted communication requests has already reached the queue size of a given queue, the communication request fails.

> *User advice:* Return value GASPI_SUCCESS does not mean, that the notification has been transferred or that the notification has arrived at the remote side. ⌟
>
> *Implementor advice:* gaspi_notify is semantically equivalent to an update of a local internally managed notification buffer with a subsequent gaspi_write. However, it should be implemented more efficiently, if supported by the network infrastructure. ⌟

9.3.3 gaspi_notify_waitsome

For the procedures with notification, gaspi_notify and the extendend function gaspi_write_notify, gaspi_write_waitsome is the correspondent wait procedure for the receiver (notified) side. gaspi_notify_waitsome is a *synchronous, non-local time-based blocking* procedure.

```
gaspi_return_t
gaspi_notify_waitsome ( gaspi_notification_id_t notification_begin
                      , gaspi_number_t notification_num
                      , gaspi_timeout_t timeout
                      )
```

Parameter:

(in) notification_ begin: the local notification ID for the first notification to wait for

(in) notification_ num: the number of notification ID's to wait for

(in) timeout: the timeout

Execution phase:

Working

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_TIMEOUT`: operation has run into timeout

`GASPI_ERROR`: operation has finished with an error ⌟

`gaspi_notify_waitsome` waits that at least one of a number of consecutive notifications residing in the local internal buffer has a value that is not zero.

The notification buffer is specified by the pair *notification_begin*, *notification_num*. It contains *notification_num* many consecutive notifications beginning at the notification with ID *notification_begin*.

If *notification_num* == 0 then `gaspi_notify_waitsome` returns immediately with `GASPI_SUCCESS`.

After successful procedure completion, i. e. return value `GASPI_SUCCESS`, the value of at least one of the notifications in the notification buffer has changed to a value that is not zero. All threads that are waiting for the notifications are notified.

If the procedure returns with `GASPI_TIMEOUT`, no notification has changed during the given period of time.

In case of an error, i. e. `GASPI_ERROR`, the values of the notifications are undefined.

> *User advice:* One scenario for the usage of `gaspi_notify_waitsome` inspecting only one notification is the following: The remote side uses a `gaspi_write` call followed by a subsequent call of `gaspi_notify` posted to the same queue and the same destination rank. Since the order of communication requests on the remote side is preserved, one can assume, that if the notification has arrived on the remote process, then the previously posted request carrying the work load have arrived. ⌟

> *User advice:* If in a multi-threaded application more than one thread calls `gaspi_notify_waitsome` for the range of notifications, then all waiting threads are notified about the change of at least one of the notifications. By inspecting the actual values of each of the notifications with `gaspi_notify_reset`, only one thread per changed notification receives a value different from zero. ⌟

> *User advice:* In a multi-threaded application the code in listing 9 selects one thread to act on the change of a single notification. The code waits in a blocking manner and thus cannot be used in fault tolerant applications. ⌟

Listing 9: Blocking waitsome in a multi-threaded application

```
#include <GASPI.h>
#include <success_or_die.h>

extern void process ( const gaspi_notification_id_t id
                    , const gaspi_notification_t val
                    );

void blocking_waitsome ( const gaspi_notification_id_t id_begin
                       , const gaspi_notification_id_t id_end
                       )
{
  ASSERT ( gaspi_notify_waitsome ( id_begin
                                 , id_end - id_begin
                                 , GASPI_BLOCK
                                 )
         );

  for (gaspi_notification_id_t i = id_begin; i < id_end; ++i)
    {
      gaspi_notification_t val = 0;

      // atomic reset
      ASSERT (gaspi_notify_reset (i, &val));

      // re-check, other threads are notified too!
      if (val != 0)
        {
          process (i, val);
        }

    }
}
```

9.3.4 gaspi_notify_reset

For the `gaspi_notify_waitsome` procedure, there is a notification initialization procedure which resets the given notification to zero. It is a *synchronous local blocking* procedure.

```
gaspi_return_t
gaspi_notify_reset ( gaspi_notification_id_t notification_id
                   , gaspi_notification_t old_notification_val
                   )
```

Parameter:

(in) notification_ id: the local notification ID to reset

(out) old_ notification_ val: notification value before reset

Execution phase:

Working

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_ERROR`: operation has finished with an error ⌟

`gaspi_notify_reset` resets the notification with ID *notification_ id* to zero. The function `gaspi_notify_reset` is an atomic operation: Threads can use `gaspi_notify_reset` to safely extract the value of a specific notification.

The notification buffer on the local side is specified by the notification ID *notification_ id*.

After successful procedure completion, i. e. return value `GASPI_SUCCESS`, the value of the notification buffer was set to zero and *old_ notification_ val* contains the content of the notification buffer before it was set to zero. To read the old value and to set the value to zero is a single atomic operation.

`gaspi_notify_reset` calls may be posted from every thread of the GASPI process.

In case of error, i. e. return value `GASPI_ERROR`, the value of *old_ notification_ val* is undefined.

9.4 Extended communication calls

All restrictions applying to `gaspi_write` and `gaspi_notify` also apply here. In case of timeout or error, no assumptions may be made regarding either the written data or the notification.

9.4.1 `gaspi_write_notify`

The `gaspi_write_notify` variant extends the simple `gaspi_write` with a notification on the remote side. This applies to communication patterns that require tighter synchronisation on data movement. The remote receiver of the data is notified when the write is finished and can verify this through the respective wait procedure. It is an *asynchronous non-local time-based blocking* procedure.

```
gaspi_return_t
gaspi_write_notify ( gaspi_segment_id_t segment_id_local
                   , gaspi_offset_t offset_local
                   , gaspi_rank_t rank
                   , gaspi_segment_id_t segment_id_remote
                   , gaspi_offset_t offset_remote
                   , gaspi_size_t size
                   , gaspi_notification_id_t notification_id
                   , gaspi_notification_t notification_value
                   , gaspi_queue_id_t queue
                   , gaspi_timeout_t timeout
                   )
```

Parameter:

(in) segment_id_local: the local segment ID to read from

(in) offset_local: the local offset in bytes to read from

(in) rank: the remote rank to write to

(in) segment_id_remote: the remote segment to write to

(in) offset_remote: the remote offset to write to

(in) size: the size of the data to write

(in) notification_id: the remote notification ID

(in) notification_value: the value of the notification to write

(in) queue: the queue to use

(in) timeout: the timeout

Execution phase:

Working

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_TIMEOUT`: operation has run into timeout

`GASPI_ERROR`: operation has finished with an error ⌟

Two new entries are inserted into the given queue.

> *Implementor advice:* The procedure is semantically equivalent to a call to `gaspi_write` and a subsequent call of `gaspi_notify`. However, it should be implemented more efficiently, if supported by the network infrastructure. ⌟

9.4.2 `gaspi_write_list`

The `gaspi_write_list` variant allows strided communication where a list of different data locations are processed at once. Semantically, it is equivalent to a sequence of calls to `gaspi_write` but it should (if possible) be more efficient. It is an *asynchronous non-local time-based blocking* procedure.

```
gaspi_return_t
gaspi_write_list ( gaspi_number_t num
                 , gaspi_segment_id_t segment_id_local[num]
                 , gaspi_offset_t offset_local[num]
                 , gaspi_rank_t rank
                 , gaspi_segment_id_t segment_id_remote[num]
                 , gaspi_offset_t offset_remote[num]
                 , gaspi_size_t size[num]
                 , gaspi_queue_id_t queue
                 , gaspi_timeout_t timeout
                 )
```

Parameter:

(in) num: the number of elements to write

(in) segment_id_local[num]: list of local segment ID's to read from

(in) offset_local[num]: list of local offsets in bytes to read from

(in) rank: the remote rank to write to

(in) segment_id_remote[num]: list of remote segments to write to

(in) offset_remote[num]: list of remote offsets to write to

(in) size[num]: list of sizes of the data to write

(in) queue: the queue to use

(in) timeout: the timeout

Execution phase:

Working

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_TIMEOUT`: operation has run into timeout

`GASPI_ERROR`: operation has finished with an error ⌟

num new entries are inserted in the given queue.

Implementor advice: The procedure is semantically equivalent to *num* subsequent calls of `gaspi_write` with the given local and remote location specification, provided that the destination rank and the used queue are invariant. However, it should be implemented more efficiently, if supported by the network infrastructure. ⌟

9.4.3 `gaspi_write_list_notify`

The `gaspi_write_list_notify` operation performs strided communication as `gaspi_write_list` but also includes a notification that the remote receiver can use to ensure that the communication step is completed. It is an *asynchronous non-local time-based blocking* procedure.

```
gaspi_return_t
gaspi_write_list_notify
                ( gaspi_number_t num
                , gaspi_segment_id_t segment_id_local[num]
                , gaspi_offset_t offset_local[num]
                , gaspi_rank_t rank
                , gaspi_segment_id_t segment_id_remote[num]
                , gaspi_offset_t offset_remote[num]
                , gaspi_size_t size[num]
                , gaspi_notification_id_t notification_id
                , gaspi_notification_t notification_value
                , gaspi_queue_id_t queue
                , gaspi_timeout_t timeout
                )
```

Parameter:

(in) num: the number of elements to write

(in) segment_id_local[num]: list of local segment ID's to read from

(in) offset_local[num]: list of local offsets in bytes to read from

(in) rank: the remote rank to be write to

(in) segment_id_remote[num]: list of remote segments to write to

(in) offset_remote[num]: list of remote offsets to write to

(in) size[num]: list of sizes of the data to write

(in) notification_id: the remote notification ID

(in) notification_value: the value of the notification to write

(in) queue: the queue to use

(in) timeout: the timeout

Execution phase:

Working

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_TIMEOUT`: operation has run into timeout

`GASPI_ERROR`: operation has finished with an error ⌟

num+1 new entries are inserted into the given queue.

> *Implementor advice:* The procedure is semantically equivalent to a call to `gaspi_write_list` and a subsequent call of `gaspi_notify`. However, it should be implemented more efficiently, if supported by the network infrastructure. ⌟

9.4.4 `gaspi_read_list`

The `gaspi_read_list` variant allows strided communication where a list of different data locations are processed at once. Semantically, it is equivalent to a sequence of calls to `gaspi_read` but it should (if possible) be more efficient. It is an *asynchronous non-local time-based blocking* procedure.

```
gaspi_return_t
gaspi_read_list ( gaspi_number_t num
                , gaspi_segment_id_t segment_id_local[num]
                , gaspi_offset_t offset_local[num]
                , gaspi_rank_t rank
                , gaspi_segment_id_t segment_id_remote[num]
                , gaspi_offset_t offset_remote[num]
                , gaspi_size_t size[num]
                , gaspi_queue_id_t queue
                , gaspi_timeout_t timeout
                )
```

Parameter:

(in) num: the number of elements to read

(in) segment_id_local[num]: list of local segment ID's to write to

(in) offset_local[num]: list of local offsets in bytes to write to

(in) rank: the remote rank to read from

(in) segment_id_remote[num]: list of remote segments to read from

(in) offset_remote[num]: list of remote offsets to read from

(in) size[num]: list of sizes of the data to read

(in) queue: the queue to use

(in) timeout: the timeout

Execution phase:

Working

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_TIMEOUT`: operation has run into timeout

`GASPI_ERROR`: operation has finished with an error ⌟

num new entries are inserted into the given queue.

9.5 Communication utilities

9.5.1 `gaspi_queue_size`

The `gaspi_queue_size` procedure is a *synchronous local blocking* procedure which determines the number of open communication requests posted to a given queue.

```
gaspi_return_t
gaspi_queue_size ( gaspi_queue_id_t queue
                 , gaspi_number_t queue_size
                 )
```

Parameter:

(in) queue: the queue to probe

(out) queue_size: the number of open requests posted to the queue

Execution phase:

Working

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_ERROR`: operation has finished with an error ⌟

After successful procedure completion, i. e. return value `GASPI_SUCCESS`, the parameter *queue_size* contains the number of open requests posted to the queue *queue*. In a threaded program this result is uncertain, since another thread may have posted an additional request in the meantime or issued a wait call.

The queue size is set to zero by a successful call to `gaspi_wait`.

In case of error, the return value is `GASPI_ERROR`. The parameter *queue_size* has an undefined value.

9.5.2 gaspi_queue_purge

The `gaspi_queue_purge` procedure is a *synchronous local time-based blocking* procedure which purges a given queue.

```
gaspi_return_t
gaspi_queue_purge ( gaspi_queue_id_t queue
                  , gaspi_timeout_t timeout
                  )
```

Parameter:

(in) queue: the queue to purge

(in) timeout: the timeout

Execution phase:

Working

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_TIMEOUT`: operation has run into timeout

`GASPI_ERROR`: operation has finished with an error ⌟

This procedure should only be invoked in the situation in which a node failure is detected by inspecting the global health state with `gaspi_state_vec_get`.

After successful procedure completion, i. e. return value `GASPI_SUCCESS`, the communication *queue* is purged. All communication requests posted to the queue *queue* are eliminated from the queue. The local GASPI process has no information about the completion of communication requests posted to the given queue since the last invocation of `gaspi_wait`.

If the procedure returns with `GASPI_TIMEOUT`, the purge request could not be completed during the given timeout. This might happen if there is another thread in a `gaspi_wait` for the same queue. A subsequent call of `gaspi_queue_purge` has to be invoked in order to complete the call.

If the procedure returns with `GASPI_ERROR`, the purge request aborted abnormally.

10 Passive communication

10.1 Introduction and overview

Passive communication has a two-sided semantic, where there is a matching receiver to a send request. Passive communication aims at communication patterns where the sender is unknown (i. e. it can be any process from the receiver perspective) but there is potentially the need for synchronisation between processes. Typical example uses cases are:

- Distributed update where many processes contribute to the data of one process.
- Pass arguments and results.
- Global error handling.

The implementation should try to enforce fairness in communication that is, no sender should see its communication request delayed indefinitely.

The passive keyword means that the communication calls should avoid busy-waiting and consume no CPU cycles, freeing the system for computation.

The send request is asynchronous, whereas the matching receive is *time-based blocking*. Due to the asynchronous nature of the send request, a complete send involves two procedure calls. First, one call which initiates the communication. This call posts a communication request to the underlying network infrastructure. The second call waits for the completion of the communication request.

A valid passive communication request requires that the local and the remote segment are allocated and that there is a connection between the local and the remote GASPI process. Otherwise, the communication request is invalid and the procedure returns with `GASPI_ERROR`.

10.2 Passive communication calls

10.2.1 `gaspi_passive_send`

The non-blocking `gaspi_passive_send` is one of the routines called by the sender side to engage in passive communication. It is an *asynchronous non-local time-based blocking* procedure.

```
gaspi_return_t
gaspi_passive_send ( gaspi_segment_id_t segment_id_local
                   , gaspi_offset_t offset_local
                   , gaspi_rank_t rank
                   , gaspi_size_t size
                   , gaspi_tag_t tag
                   , gaspi_timeout_t timeout
                   )
```

Parameter:

(in) segment_id_local: the local segment ID from which the data is sent

(in) offset_local: the local offset from which the data is sent

(in) rank: the remote rank to which the data is sent

(in) size: the size of the data to be sent

(in) tag: the tag to be sent with the data

(in) timeout: the timeout

Execution phase:

Working

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_TIMEOUT`: operation has run into timeout

`GASPI_ERROR`: operation has finished with an error ⌟

`gaspi_passive_send` posts a passive communication request which asynchronously transfers a contiguous block of *size* bytes from a source location of the local GASPI process to the remote GASPI process with the indicated rank *rank*. On the remote side, a corresponding `gaspi_passive_receive` has to be posted. The source location is specified by the pair *segment_id_local*, *offset_local*.

There is a size limit for the data sent with `gaspi_passive_send`. The maximum size is returned by the function `gaspi_passive_transfer_size_max`.

The passive data *tag* can be used to discriminate different types of data.

A valid `gaspi_send` communication request requires that the local and the remote segment are allocated and that there is a connection between the local and the remote GASPI process. Otherwise, the communication request is invalid and the procedure returns with `GASPI_ERROR`.

After successful procedure completion, i. e. return value `GASPI_SUCCESS`, the passive communication request has been posted to the underlying network infrastructure. One new entry is inserted into the passive queue.

Successive `gaspi_passive_send` calls posted to the same destination rank are non-overtaking. Non-overtaking means that the order of communication requests is preserved on the remote side. In particular, one can assume, that if the data from the later request has arrived on the remote process, also the data from the earlier posted request to the same process have arrived on the remote side.

`gaspi_passive_send` calls may be posted from every thread of the GASPI process.

If the procedure returns with `GASPI_TIMEOUT`, the communication request could not be posted to the hardware during the given timeout. This can happen, if another thread is in a `gaspi_passive_wait`. A subsequent call of `gaspi_passive_send` needs to be invoked in order to complete the procedure call.

A passive communication request can be considered as completed, if the the correspondent `gaspi_passive_wait` returns with `GASPI_SUCCESS`.

If the passive communication queue is full at the time when a new passive communication request is posted, i. e. the number of posted communication requests has already reached the queue size, the communication request fails and the procedure returns with return value GASPI_ERROR. If a saturated passive communication queue is detected one has to invoke a gaspi_passive_wait in order to wait for all the posted requests to be finished.

> *User advice:* The parameter *tag* is *not* used to match sends and reveices. It is just an additional information that the sender attaches to the data, e. g. the type. On the receiver side, *all* messages are accepted, regardless of the value of the tag. However, by reading the submitted tag-value, the receiver can act differently on different types of data. ⌟

> *User advice:[see also the advice in 9.2.1 on page 50]* Return value GASPI_SUCCESS does not mean, that the data has been transferred or buffered or that the data has arrived at the remote side.
> It is allowed to write data to the source location while the communication is ongoing. However, the result on the remote side would be some undefined interleaving of the data that was present when the call was issued and the data that was written later.
> It is also allowed to read from the source location while the communcation is ongoing and such a read would retrieve the data written by the application. ⌟

> *User advice:* If the parameter *build_ infrastructure* is not set, a connection has to be established between the processes before the gaspi_passive_send can be be used. This is accomplished calling the procedure gaspi_connect. ⌟

10.2.2 gaspi_passive_wait

The gaspi_passive_wait procedure is an *asynchronous non-local time-based blocking* procedure which waits until all posted passive send requests are processed by the network infrastructure.

```
gaspi_return_t
gaspi_passive_wait (gaspi_timeout_t timeout)
```

Parameter:

(in) Timeout: the timeout

Execution phase:

Working

Return values:

GASPI_SUCCESS: operation has returned successfully

GASPI_TIMEOUT: operation has run into timeout

GASPI_ERROR: operation has finished with an error ⌟

The passive wait operation makes sure all passive send requests are finished.

After successful procedure completion, i. e. return value `GASPI_SUCCESS`, the hitherto posted passive communication requests have been processed by the network infrastructure. After that, any passive communication request which has been posted can be considered as completed on the local side.

`gaspi_passive_wait` procedure calls may be posted from every thread of the local GASPI process. However, the wait operation is a thread exclusive operation and therefore needs privileged access to the queue which means that if a passive send is done while a wait is in operation, the passive send blocks to ensure correctness. Enforcing this provides correctness and safety to the user while being easier for the implementor and still allows for a high performance implementation. As a consequence, successive `gaspi_passive_wait` calls invoked for the same queue by different threads are processed in some sequence one after another.

If the procedure returns with `GASPI_TIMEOUT`, the wait request could not be completed during the given timeout. This can happen, if there are still some passive communication requests to be completed or if there is another thread in a `gaspi_passive_wait` for the same queue.

If the procedure returns with `GASPI_ERROR`, the wait request aborted abnormally.

In both cases, `GASPI_TIMEOUT` and `GASPI_ERROR`, the GASPI state vector should be checked in order to eliminate the possibility of a failure. If a failure is detected, all of the `gaspi_passive_send` requests which have been posted to the given queue since the last `gaspi_passive_wait` are in an undefined state. Here, undefined state means that the local GASPI process does not know which requests have been processed and which requests are still outstanding. A call to `gaspi_passive_queue_purge` has to be invoked in order to reset the passive queue.

> *User advice:[identical to the advice in 9.2.3 on page 53]* Return value `GASPI_SUCCESS` means, that the data of all posted passive communication requests has been transferred to the remote side. It does not mean, that the data has arrived at the remote side. However, write accesses to the local source location will not affect the data that is placed in the remote target location. ⌟

10.2.3 `gaspi_passive_receive`

The *synchronous non-local time-based blocking* `gaspi_passive_receive` is one of the routines called by the receiver side to engage in passive communication.

```
gaspi_return_t
gaspi_passive_receive ( gaspi_segment_id_t segment_id_local
                      , gaspi_offset_t offset_local
                      , gaspi_rank_t rank
                      , gaspi_tag_t tag
                      , gaspi_timeout_t timeout
                      )
```

Parameter:

(in) segment_id_local: the local segment ID where to write the data

(in) offset_local: the local offset where to write the data

(out) rank: the remote rank from which the data is transferred

(out) tag: the tag which has been sent with the data

(in) timeout: the timeout

Execution phase:

Working

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_TIMEOUT`: operation has run into timeout

`GASPI_ERROR`: operation has finished with an error ⌟

`gaspi_passive_receive` receives a contiguous block of data into a target location from some unspecified remote GASPI process. The target location is specified by the pair *segment_id_local*, *offset_local*.

There is no need for the `gaspi_passive_receive` procedure to be active before a corresponding `gaspi_passive_send` procedure is invoked. However, as long as there is no matching receive, the `gaspi_passive_send` cannot achieve any progress and a `gaspi_passive_wait` that includes such a `gaspi_passive_send` cannot return `GASPI_SUCCESS`.

The target location needs to have enough space to hold the maximum passive transfer size that could be sent be any other process. Otherwise, the received data might overwrite memory regions outside of the allocated memory and the application will be in an undefined state.

A valid `gaspi_passive_receive` communication request requires that the local destination segment is allocated and that there is a connection between the local and the remote GASPI process from which a data transfer originates. Otherwise, the communication request is invalid and the procedure returns with `GASPI_ERROR`.

After successful procedure completion, i. e. return value `GASPI_SUCCESS`, the data has been received and is available at the target location. Further *rank* contains the rank of the sending process and *tag* contains the tag associated to the communication request.

Successive `gaspi_passive_receive` calls posted by two different threads using two different target locations are allowed. However, either the first incoming data is received by the first thread or the by the second. That means that the `gaspi_passive_receive` should be posted only from a single thread of a GASPI process.

If the procedure returns with `GASPI_TIMEOUT`, there was no pending communication request in the queue. The output parameters *rank* and *tag* have no defined value.

> *User advice:* It is allowed to write data to the local target location while the passive communication is ongoing. However, the content of the memory would be some undefined interleaving of the data transferred from remote side and the data written locally.
> Also, it is allowed to read from the local target location while the passive communication is ongoing. Such a read would retrieve some undefined interleaving of the data that was present when the call was issued and the data that was transferred from the remote side. ⌟

> *Implementor advice:* A quality implementation enforces fairness in communication that is, no sender should see its communication request delayed indedinitely. The passive keyword means the communication calls shall avoid busy-waiting and consume no CPU cycles, freeing the system for computation. ⌟

10.3 Passive communication utilities

10.3.1 `gaspi_passive_queue_size`

The `gaspi_passive_queue_size` procedure is a *synchronous local blocking* procedure which determines the number of communication requests posted to the passive queue.

```
gaspi_return_t
gaspi_passive_queue_size (gaspi_number_t queue_size)
```

Parameter:

(out) queue_size: Number of requests posted to the passive communication queue

Execution phase:

Working

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_ERROR`: operation has finished with an error ⌟

After successful procedure completion, i. e. return value `GASPI_SUCCESS`, the parameter *queue_ size* contains the number of posted requests. In a threaded program this result is uncertain, since another thread may have posted an additional request or a wait call in the meantime.

The queue size is set to zero by a successful call to `gaspi_passive_wait`.

In case of error, the return value is `GASPI_ERROR`. The value of *passive_ queue_ size* is undefined.

10.3.2 `gaspi_passive_queue_purge`

The `gaspi_passive_queue_purge` procedure is a *synchronous local time-based blocking* procedure which purges the passive queue.

```
gaspi_return_t
gaspi_passive_queue_purge (gaspi_timeout_t timeout)
```

Parameter:

(in) timeout: the timeout

Execution phase:

Working

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_TIMEOUT`: operation has run into timeout

`GASPI_ERROR`: operation has finished with an error ⌟

This procedure should only be invoked in the situation in which a node failure is detected by inspecting the global health state with `gaspi_state_vec_get`.

After successful procedure completion, i. e. return value `GASPI_SUCCESS`, the passive communication queue is purged. All communication requests posted to the passive queue are eliminated. The local GASPI process has no information about the completion of communication requests posted to the passive queue since the last invocation of `gaspi_passive_wait`.

If the procedure returns with `GASPI_TIMEOUT`, the purge request could not be completed during the given timeout. This might happen if there is another thread in a `gaspi_passive_wait`. A subsequent call of `gaspi_passive_queue_purge` has to be invoked in order to complete the call.

If the procedure returns with `GASPI_ERROR`, the purge request aborted abnormally.

11 Global atomics

11.1 Introduction and Overview

Atomic counters are globally accessible variables to which atomic access is provided through three operations: set, fetch-and-add and compare-and-swap.

An atomic operation is an operation which is guaranteed to be executed without fear of interference from other processes during the procedure call. Only one GASPI process at a time has access to the global variable and can modify it.

Atomic operations are also guaranteed to be fair. That means no GASPI process should see its atomic operation request delayed indefinitely.

The number of atomic counters available can be defined by the user through the configuration structure at start-up and cannot be changed during run-time. The maximum number of available atomic counters is implementation dependent.

> *Implementor advice:* The distribution of the counters should be divided evenly over the available nodes to avoid too much contention on a single node holding the atomic counters. ⌟

11.2 Atomic operation calls

11.2.1 gaspi_counter_set

The `gaspi_counter_set` procedure is a *synchronous non-local time-based blocking* procedure which atomically set the value of a global counter.

```
gaspi_return_t
gaspi_counter_set ( gaspi_counter_id_t counter_id
                  , gaspi_counter_value_t value
                  , gaspi_timeout_t timeout
                  )
```

Parameter:

(in) counter_id: the counter to set

(in) value: the value to which the global atomic counter is set

(in) timeout: the timeout

Execution phase:

Working

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_TIMEOUT`: operation has run into timeout

`GASPI_ERROR`: operation has finished with an error ⌟

`gaspi_counter_set` atomically sets the value of the counter *counter_ id* to the value given by *value*.

After successful procedure completion, i. e. return value `GASPI_SUCCESS`, the value of the atomic counter was set to *value*.

If the procedure returns with `GASPI_TIMEOUT`, the reset request could not be completed during the given timeout. A subsequent call of `gaspi_counter_set` needs to be invoked in order to complete the operation.

If the procedure returns with `GASPI_ERROR`, the reset request aborted abnormally and the counter has an undefined value.

In both cases, `GASPI_TIMEOUT` and `GASPI_ERROR`, the GASPI state vector should be checked in order to deal with possibile failures.

11.2.2 `gaspi_counter_fetch_add`

The `gaspi_counter_fetch_add` procedure is a *synchronous non-local time-based blocking* procedure which atomically adds a given value to a global counter.

```
gaspi_return_t
gaspi_counter_fetch_add ( gaspi_counter_id_t counter_id
                        , gaspi_counter_value_t value_add
                        , gaspi_counter_value_t value_old
                        , gaspi_timeout_t timeout
                        )
```

Parameter:

(in) counter_ id: the global atomic counter to be used for the operation

(in) value_ add: the value which is to be added to the global atomic counter

(out) value_ old: the value of the counter before the operation

(in) timeout: the timeout

Execution phase:

Working

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_ERROR`: operation has finished with an error ⌟

`gaspi_counter_fetch_add` atomically adds the value of *value_ add* to the value of the counter *counter_ id*.

After successful procedure completion, i. e. return value `GASPI_SUCCESS`, the parameter *value_ old* contains the value of the atomic counter before the operation has been applied.

If the procedure returns with `GASPI_TIMEOUT`, the fetch and add request could not be completed during the given timeout. The parameter *value_ old* has an undefined value. A subsequent call of `gaspi_counter_fetch_add` needs to be invoked in order to complete the operation.

If the procedure returns with `GASPI_ERROR`, the fetch and add request aborted abnormally. The parameter *value_ old* as well as the counter have undefined values.

In both cases, `GASPI_TIMEOUT` and `GASPI_ERROR`, the GASPI state vector should be checked in order to deal with possible failures.

11.2.3 `gaspi_counter_compare_swap`

The `gaspi_counter_compare_swap` procedure is a *synchronous non-local time-based blocking* procedure which atomically compares the value of a global counter against some user given value and in case these values are equal replaces the old counter value by a new value.

```
gaspi_return_t
gaspi_counter_compare_swap ( gaspi_counter_id_t counter_id
                           , gaspi_counter_value_t comparator
                           , gaspi_counter_value_t value_new
                           , gaspi_counter_value_t value_old
                           , gaspi_timeout_t timeout
                           )
```

Parameter:

(in) counter_ id: the global atomic counter to be used for the operation

(in) comparator: the value which is compared to the value of the counter

(in) value_ new: the new value to which the counter is set if the result of the comparison is true

(out) value_ old: the value of the counter before the operation

(in) timeout: the timeout

Execution phase:

Working

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_ERROR`: operation has finished with an error ⌟

gaspi_counter_compare_swap atomically compares the value of the counter *counter_ id* to the value of *comparator*. If the comparison is true, the counter is set to *value_ new*. If the comparison is false, the counter keeps its value.

After successful procedure completion, i. e. return value GASPI_SUCCESS, the parameter *value_ old* contains the value of the atomic counter before the comparision was done.

If the procedure returns with GASPI_TIMEOUT, the compare and swap request could not be completed during the given timeout. The parameter *value_ old* has an undefined value. A subsequent call of gaspi_counter_compare_swap needs to be invoked in order to complete the operation.

If the procedure returns with GASPI_ERROR, the compare and swap request aborted abnormally. The parameter *value_ old* as well as the counter have an undefined value.

In both cases, GASPI_TIMEOUT and GASPI_ERROR, the GASPI state vector should be checked in order to deal with possible failures.

11.2.4 Examples

The example in listing 10 illustrates the usage of global atomic counters for implementing a global resource lock. The example is implemented with timeout.

Listing 10: GASPI global resource lock implemented with atomic counters

```
#include <GASPI.h>
#include <assert.h>

#define SUCCESS_OR_RETURN(f)                    \
  {                                             \
    const int ec = (f);                         \
                                                \
    if (ec != GASPI_SUCCESS)                    \
      {                                         \
        return ec;                              \
      }                                         \
  }

#define VAL_UNLOCKED -1
#define VAL_NONE -2

gaspi_return_t
global_lock_init ( const gaspi_counter_id_t lock_cnt_id
                 , const gaspi_timeout_t timeout
                 )
{
  static unsigned short phase = 0;

  gaspi_timeout_t time_left = timeout;

  switch (phase)
    {
    case 0:
      {
        gaspi_rank_t iProc = GASPI_NORANK;

        SUCCESS_OR_RETURN (gaspi_proc_rank (&iProc));

        if (0 == iProc)
          {
            gaspi_time_t time_start = GASPI_NOTIME;
            gaspi_time_t time_end = GASPI_NOTIME;

            SUCCESS_OR_RETURN (gaspi_time_get (&time_start));
            SUCCESS_OR_RETURN (gaspi_counter_reset ( lock_cnt_id
                                                   , VAL_UNLOCKED
                                                   , time_left
                                                   )
                              );
            SUCCESS_OR_RETURN (gaspi_time_get (&time_end));

            if (timeout >= 0)
              {
                time_left -= time_end - time_start;
              }
          }

        phase = 1;
      }
    case 1:
      if (timeout >= 0 && time_left < 0)
        {
          return GASPI_TIMEOUT;
        }

      SUCCESS_OR_RETURN (gaspi_barrier ( GASPI_GROUP_ALL
                                       , time_left
                                       )
                        );

      phase = 0;

    default:
      assert (false);
    }

  return GASPI_SUCCESS;
}

gaspi_return_t
```

```
global_try_lock ( const gaspi_counter_id_t lock_cnt_id
                , const gaspi_timeout_t timeout
                )
{
  gaspi_rank_t iProc = GASPI_NORANK;

  SUCCESS_OR_RETURN (gaspi_proc_rank (&iProc));

  gaspi_counter_value_t current_value = VAL_NONE;

  SUCCESS_OR_RETURN (gaspi_counter_compare_swap ( lock_cnt_id
                                                , VALUE_UNLOCKED
                                                , iProc
                                                , &current_value
                                                , timeout
                                                )
                    );

  return (current_value == VALUE_UNLOCKED) ? GASPI_SUCCESS
                                           : GASPI_ERROR
                                           ;
}

gaspi_return_t
global_unlock ( const gaspi_counter_id_t lock_cnt_id
              , const gaspi_timeout_t timeout
              )
{
  gaspi_rank_t iProc = GASPI_NORANK;

  SUCCESS_OR_RETURN (gaspi_proc_rank (&iProc));

  gaspi_counter_value_t current_value = VAL_NONE;

  SUCCESS_OR_RETURN (gaspi_counter_compare_swap ( lock_cnt_id
                                                , iProc
                                                , VALUE_UNLOCKED
                                                , &current_value
                                                , timeout
                                                )
                    );

  assert (current_value == iProc);

  return GASPI_SUCCESS;
}
```

12 Collective communication

12.1 Introduction and overview

Collective operations are collective with respect to a given group. A necessary condition for successful collective procedure completion is that all GASPI processes forming the given group have invoked the operation.

Collective operations support both synchronous and asynchronous implementations as well as time-based blocking. That means, progress towards successful procedure completion can be achieved either inside the call (for a synchronous implementation) or outside of the call (for an asynchronous implementation) before the procedure exits. In the case of a timeout (which is indicated by return value `GASPI_TIMEOUT`) the operation is then continued in the next call of the procedure. This implies that a collective operation may involve several procedure calls until completion. Completion is indicated by return value `GASPI_SUCCESS`.

Collective operations are exclusive per group, i. e. only one collective operation of a specific type on a given group can run at a given time. Starting a specific collective operation before another one of the same kind is not finished on all processes of the group (and marked as such) is not allowed and yields undefined behavior. For example, two allreduce operations for one group can not run at the same time; however, an allreduce and a barrier operation can run at the same time.

The timeout is a necessary condition in order to be able to write failure tolerant code. Timeout = 0 makes an atomic portion of progress in the operation if possible. If progress is possible, the procedure returns as soon as the atomic portion of progress is achieved. Otherwise, the procedure returns immediately. Here, an atomic portion of progress is defined as the smallest set of non-dividable instructions in the current state of the collective operation.

Reduction operations can be defined by the application via callback functions.

User advice: Not every collective operation will be implementable in an asynchronous fashion – for example if a user-defined callback function is used within a global reduction. Progress in this case can only be achieved inside of the call. Especially for large systems this implies that a collective potentially has to be called a substantial number of times in order to complete – especially if used in combination with `GASPI_TEST`. In this combination the called collective immediately returns (after completing local work) and never waits for data from remote processes. A corresponding code fragment in this case would assume the form:

```
while (GASPI_allreduce_user ( buffer_send
                             , buffer_receive
                             , char num
                             , size_element
                             , reduce_operation
                             , reduce_state
                             , group
                             , GASPI_TEST
                             ) != GASPI_SUCCESS
      )
{
      work();
}
```

⌟

12.2 Barrier synchronisation

12.2.1 gaspi_barrier

The `gaspi_barrier` procedure is a *collective time-based blocking* procedure. An implementation is free to provide it as a synchronous or an asynchronous procedure.

```
gaspi_return_t
gaspi_barrier ( gaspi_group_t group
              , gaspi_timeout_t timeout
              )
```

Parameter:

(in) group: the group of ranks which should participate in the barrier

(in) timeout: the timeout

Execution phase:

Working

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_TIMEOUT`: operation has run into timeout

`GASPI_ERROR`: operation has finished with an error ⌟

`gaspi_barrier` blocks the caller until all group members of *group* have invoked the procedure or if *timeout* milliseconds have been reached since procedure invocation. After successful procedure completion, i. e. return value `GASPI_SUCCESS`, all group members have invoked the procedure. In case of `GASPI_TIMEOUT` it is unknown whether or not all Gaspi processes forming the given group have invoked the call.

Progress towards successful `gaspi_barrier` completion is achieved even if the procedure exits due to timeout. The barrier is then continued in the next call of the procedure. This implies that a barrier operation may involve several `gaspi_barrier` calls until completion.

Barrier operations are exclusive per group, i. e. only one barrier operation on a given group can run at a time. Starting a barrier operation in another thread before a previously invoked barrier is finished on all processes of the group is not allowed and yields undefined behavior.

In case of error, the return value is `GASPI_ERROR`. The error vector should be investigated.

> *User advice:* The barrier is supposed to synchronise processes and not threads. ⌟

12.2.2 Examples

In the following example a `gaspi_barrier` is interrupted after 100 ms in order to check for errors.

```
gaspi_return_t err;

do
  {
    err = gaspi_barrier (g, 100);

    if (err == GASPI_TIMEOUT && error vector indicates error)
      {
        goto ERROR_HANDLING;
      }
  }
while (err != GASPI_SUCCESS);
```

The following example shows a non-blocking barrier. Some local work (in this case: cleanup) is performed, overlapping it with the barrier and only then a full synchronisation is achieved by calling the barrier again with a blocking semantics (if needed).

```
const gaspi_return_t err = GASPI_barrier (g, GASPI_TEST);

do_local_cleanup();

if (err != GASPI_ERROR)
{
    err = GASPI_barrier (g, GASPI_BLOCK);
}
```

12.3 Predefined global reduction operations

12.3.1 `gaspi_allreduce`

The `gaspi_allreduce` procedure is a *collective time-based blocking* procedure. An implementation is free to provide it as a synchronous or an asynchronous procedure.

```
gaspi_return_t
gaspi_allreduce ( gaspi_pointer_t buffer_send
                , gaspi_pointer_t buffer_receive
                , unsigned char num
                , gaspi_operation_t operation
                , gaspi_datatype_t datatype
                , gaspi_group_t group
                , gaspi_timeout_t timeout
                )
```

Parameter:

(in) buffer_ send: pointer to the buffer where the input is placed

(in) buffer_ receive: pointer to the buffer where the result is placed

(in) num: the number of elements to be reduced on each process

(in) operation: the GASPI reduction operation type

(in) datatype: the GASPI element type

(in) group: the group of ranks which participate in the reduction operation

(in) timeout: the timeout

Execution phase:

Working

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_TIMEOUT`: operation has run into timeout

`GASPI_ERROR`: operation has finished with an error ⌟

`gaspi_allreduce` combines the *num* elements of type *datatype* residing in *buffer_send* on each process in accordance with the given *operation*. The reduction operation is on a per element basis, i. e. the operation is applied to each of the elements. `gaspi_allreduce` blocks the caller until all data is available that is needed to calculate the result or if *timeout* milliseconds have been reached since procedure invocation. After successful procedure completion, i. e. return value `GASPI_SUCCESS`, all group members have invoked the procedure and *buffer_receive* contains the result of the reduction operation on every GASPI process of *group*. In case of `GASPI_TIMEOUT` not all data is available that is needed to calculate the result.

Progress towards successful `gaspi_allreduce` completion is achieved even if the procedure exits due to timeout. The reduction operation is then continued in the next call of the procedure. This implies that a reduction operation may involve several `gaspi_allreduce` calls until completion.

Reduction operations are exclusive per group, i. e. only one reduction operation on a given group can run at a time. Starting a reduction operation for the same group in a separate thread before previously innvoked operation is finished on all processes of the group is not allowed and yields undefined behavior.

The *buffer_send* as well as the *buffer_receive* do not need to reside in the global address space. `gaspi_allreduce` copies the send buffer into an internal buffer at the first invocation. The result is copied from an internal buffer into the receive buffer immediatley before the procedure returns successfully. The buffers need to have the appropriate size to host all of the *num* elements. Otherwise the reduction operation yields undefined behavior. The maximum permissible number of elements is implementation dependent and can be retrieved by `gaspi_allreduce_elem_max`.

In case of error, the return value is `GASPI_ERROR`. The error vector should be examined. *buffer_receive* has an undefined value.

In case of `GASPI_TIMEOUT`, the reduction operation is not finished yet, i. e. not all data is available that is needed to calculate the result. The *buffer_receive* has an undefined value.

12.3.2 Predefined reduction operations

There are three predefined reduction operations:

```
typedef enum { GASPI_OP_MIN
             , GASPI_OP_MAX
             , GASPI_OP_SUM
             } gaspi_operation_t;
```

GASPI_OP_MIN determines the minimum of the elements of each column of the input vector.

GASPI_OP_MAX determines the maximum of the elements of each column of the input vector.

GASPI_OP_SUM sums up all elements of each column of the input vector.

12.3.3 Predefined types

And the types are:

```
typedef enum { GASPI_TYPE_INT
             , GASPI_TYPE_UINT
             , GASPI_TYPE_LONG
             , GASPI_TYPE_ULONG
             , GASPI_TYPE_FLOAT
             , GASPI_TYPE_DOUBLE
             } gaspi_datatype_t;
```

GASPI_TYPE_INT integer

GASPI_TYPE_UINT unsigned integer

GASPI_TYPE_LONG long

GASPI_TYPE_ULONG unsigned long

GASPI_TYPE_FLOAT float

GASPI_TYPE_DOUBLE double

12.4 User-defined global reduction operations

12.4.1 `gaspi_allreduce_user`

The procedure `gaspi_allreduce_user` allows the user to specify its own reduction operation. Only operations are supported which are commutative and associative. It is a *collective time-based blocking* procedure. An implementation is free to provide it as a synchronous or an asynchronous procedure.

```
gaspi_return_t
gaspi_allreduce_user ( gaspi_pointer_t buffer_send
                     , gaspi_pointer_t buffer_receive
                     , unsigned char num
                     , gaspi_size_t size_element
                     , gaspi_reduce_operation_t reduce_operation
                     , gaspi_reduce_state_t reduce_state
                     , gaspi_group_t group
                     , gaspi_timeout_t timeout
                     )
```

Parameter:

(in) buffer_send: pointer to the buffer where the input is placed

(in) buffer_receive: pointer to the buffer where the result is placed

(in) num: the number of elements to be reduced on each process

(in) size_ element: Size in bytes of one element to be reduced

(in) reduce_ operation: pointer to the user defined reduction operation procedure

(in,out) reduce_ state: reduction state vector

(in) group: the group of ranks which participate in the reduction operation

(in) timeout: the timeout

Execution phase:

Working

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_TIMEOUT`: operation has run into timeout

`GASPI_ERROR`: operation has finished with an error ⌟

`gaspi_allreduce_user` has the same semantics as the predefined reduction operation `gaspi_allreduce` described in the last section.

A user defined reduction operation *reduce_ operation* and a user defined state *reduce_ state* are passed.

The elements on which the user defined reduction operation is applied are described by their byte size *size_ element*. The entire size of the data to be reduced, i. e. *num* times *size_ element*, must not be larger than the internal buffer size of `gaspi_allreduce_user`. The internal buffer size can be queried through `gaspi_allreduce_buf_size`.

12.4.2 User defined reduction operations

The prototype for the user defined reduction operations is the following:

```
gaspi_return_t
gaspi_reduce_operation ( gaspi_pointer_t operand_one
                       , gaspi_pointer_t operand_two
                       , gaspi_pointer_t result
                       , gaspi_state_t state
                       , gaspi_timeout_t timeout
                       )
```

Parameter:

(in) operand_ one: pointer to the first operand

(in) operand_ two: pointer to the second operand

(in) result: pointer to the result

(in) state: pointer to the state

(in) timeout: the timeout

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_TIMEOUT`: operation has run into timeout

`GASPI_ERROR`: operation has finished with an error ⌟

A pointer to the first operand and a pointer to the second operand are passed. The result is stored in the memory represented by the pointer *result*. In addition to the actual data, also a state can be passed to the operator which might be required in order to compute the result. In order to meet real time system specifications, a timeout can be passed to the user defined reduction operator. The reduction operator should return a `gaspi_return_t` with the same semantics, i. e. `GASPI_SUCCESS` for successful procedure completion. `GASPI_TIMEOUT` in case of timeout and `GASPI_ERROR` in case of error.

The user defined reduction operator needs to be commutative and associative.

The reduce operator type passed to `gaspi_allreduce_user` is a pointer to a function with the prototype described above.

typedef `gaspi_reduce_operation`* `gaspi_reduce_operation_t`

The Gaspi *reduction operation type* ⌟

12.4.3 allreduce state

The allreduce state type

typedef void* `gaspi_state_t`

The Gaspi *reduction operation state type* ⌟

is a pointer to a state which may be passed to the user defined reduction operation. A state may contain additional information beside the actual data to be reduced needed to perform the reduction operation.

13 Gaspi getter functions

The Gaspi specification provides getter functions for all entries in the Gaspi configuration. These getter functions are *synchronous local blocking* procedures which, after successful procedure completion (i. e. return value `GASPI_SUCCESS`), read out the corresponding value of the current configuration setting.

The values of the parameters in the Gaspi configuration are determined in `gaspi_proc_init` at startup. If the value of one of these parameters is compliant with the system capabilities, the parameter is set to the requested/preferred value. Otherwise, the parameter is set to the maximum value compliant with the system capabilities. The values of the parameters realised in the GASPI configuration are implementation specific.

In case of error, the return value is `GASPI_ERROR` and the corresponding parameter in the getter function has an undefined value.

13.1 Getter functions for group management

13.1.1 gaspi_group_max

```
gaspi_return_t
gaspi_group_max (gaspi_number_t group_max)
```

Parameter:

(out) group_ max: the total number of groups

Execution phase:

Working

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_ERROR`: operation has finished with an error ⌟

13.2 Getter functions for segment management

13.2.1 gaspi_segment_max

```
gaspi_return_t
gaspi_segment_max (gaspi_number_t segment_max)
```

Parameter:

(out) segment_ max: the total number of permissible segments

Execution phase:

Working

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_ERROR`: operation has finished with an error ⌟

13.3 Getter functions for communication management

13.3.1 gaspi_queue_num

```
gaspi_return_t
gaspi_queue_num (gaspi_number_t queue_num)
```

Parameter:

(out) queue_num: the number of available queues

Execution phase:

Working

Return values:

GASPI_SUCCESS: operation has returned successfully

GASPI_ERROR: operation has finished with an error ⌟

13.3.2 gaspi_queue_size_max

```
gaspi_return_t
gaspi_queue_size_max ( gaspi_queue_id_t queue
                     , gaspi_number_t queue_size_max
                     )
```

Parameter:

(in) queue: the queue to probe

(out) queue_size_max: the maximum number of simultaneous requests allowed

Execution phase:

Working

Return values:

GASPI_SUCCESS: operation has returned successfully

GASPI_ERROR: operation has finished with an error ⌟

13.3.3 gaspi_transfer_size_max

```
gaspi_return_t
gaspi_transfer_size_max (gaspi_size_t transfer_size_max)
```

Parameter:

(out) transfer_size_max: the maximum transfer size allowed for a single request

Execution phase:

Working

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_ERROR`: operation has finished with an error ⌟

13.3.4 `gaspi_notification_num`

```
gaspi_return_t
gaspi_notification_num (gaspi_number_t notification_num)
```

Parameter:

(out) notification_num: the number of available notifications

Execution phase:

Working

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_ERROR`: operation has finished with an error ⌟

13.4 Getter functions for passive communication

13.4.1 `gaspi_passive_queue_size_max`

```
gaspi_return_t
gaspi_passive_queue_size_max (gaspi_number_t queue_size_max)
```

Parameter:

(out) queue_size_max: maximum number of requests allowed in the passive communication queue

Execution phase:

Working

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_ERROR`: operation has finished with an error ⌟

13.4.2 `gaspi_passive_transfer_size_max`

```
gaspi_return_t
gaspi_passive_transfer_size_max (gaspi_size_t transfer_size_max)
```

Parameter:

(out) transfer_ size_ max: maximal transfer size per single passive communication request

Execution phase:

Working

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_ERROR`: operation has finished with an error ⌟

13.5 Getter functions for atomic operations

13.5.1 `gaspi_counter_num`

```
gaspi_return_t
gaspi_counter_num (gaspi_number_t counter_num)
```

Parameter:

(out) counter_ num: The number of global atomic counters available

Execution phase:

Working

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_ERROR`: operation has finished with an error ⌟

13.6 Getter functions for collective communication

13.6.1 `gaspi_allreduce_buf_size`

```
gaspi_return_t
gaspi_allreduce_buf_size (gaspi_size_t buf_size)
```

Parameter:

(out) buf_ size: the size of the internal buffer in `gaspi_allreduce_user`

Execution phase:

Working

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_ERROR`: operation has finished with an error ⌟

13.6.2 `gaspi_allreduce_elem_max`

```
gaspi_return_t
gaspi_allreduce_elem_max (gaspi_number_t elem_max)
```

Parameter:

(out) elem_max: the maximum number of elements allowed in `gaspi_allreduce`

Execution phase:

Working

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_ERROR`: operation has finished with an error ⌟

13.7 Getter functions related to infrastructure

13.7.1 `gaspi_network_type`

```
gaspi_return_t
gaspi_network_type (gaspi_network_t network_type)
```

Parameter:

(out) network_type: the chosen network type

Execution phase:

Working

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_ERROR`: operation has finished with an error ⌟

13.7.2 gaspi_build_infrastructure

```
gaspi_return_t
gaspi_build_infrastructure (gaspi_bool_t build_infrastructure)
```

Parameter:

(out) build_ infrastructure: the current value of *build_ infrastructure*

Execution phase:

Working

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_ERROR`: operation has finished with an error ⌟

14 GASPI Environmental Management

14.1 Implementation Information

14.1.1 gaspi_version

The `gaspi_version` procedure is a *synchronous local blocking* procedure which determines the version of the running GASPI installation.

```
gaspi_return_t
gaspi_version (float version)
```

Parameter:

(out) version: The version of the running GASPI installation

Execution phase:

Any

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_ERROR`: operation has finished with an error ⌟

After successful procedure completion, i. e. return value `GASPI_SUCCESS` *version* contains the version of the running GASPI installation.

In case of error, the return value is `GASPI_ERROR`. The output parameter *version* has an undefined value.

14.2 Timing information

14.2.1 gaspi_time_get

The gaspi_time_get procedure is a *synchronous local blocking* procedure which determines the time elapsed since an arbitrary point of time in the past.

```
gaspi_return_t
gaspi_time_get (gaspi_time_t wtime)
```

Parameter:

(out) wtime: time elapsed in milliseconds

Execution phase:

Working

Return values:

GASPI_SUCCESS: operation has returned successfully

GASPI_ERROR: operation has finished with an error ⌟

After successful procedure completion, i. e. return value GASPI_SUCCESS, the parameter *wtime* contains elapsed time in milliseconds since an arbitrary point in the past. The parameter *wtime* is not synchronised among the different GASPI processes.

In case of error, the return value is GASPI_ERROR. The value of the output parameter *wtime* is undefined.

14.2.2 gaspi_time_ticks

The gaspi_time_ticks procedure is a *synchronous local blocking* procedure which returns the resolution of the internal timer in terms of milliseconds.

```
gaspi_return_t
gaspi_time_ticks (gaspi_time_t resolution)
```

Parameter:

(out) resolution: the resolution of the internal timer in milliseconds

Execution phase:

Any

Return values:

GASPI_SUCCESS: operation has returned successfully

`GASPI_ERROR`: operation has finished with an error ⌟

After successful procedure completion, i. e. return value `GASPI_SUCCESS`, the parameter *resolution* contains the resolution of the internal timer in units of milliseconds.

In case of error, the return value is `GASPI_ERROR`. The value of the output parameter *resolution* is undefined.

14.3 Error Codes and Classes

14.3.1 Gaspi error codes

In principle all return values less than zero represent an error. Every implementation is free to define specific error codes.

14.3.2 `gaspi_error_message`

The `gaspi_error_message` procedure is a *synchronous local blocking* procedure which translates an error code to a text message.

```
gaspi_return_t
gaspi_error_message ( gaspi_return_t error_code
                    , gaspi_string_t error_message
                    )
```

Parameter:

(in) error_ code: the error code to be translated

(out) error_ message: the error message

Execution phase:

Any

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_ERROR`: operation has finished with an error ⌟

After successful procedure completion, i. e. return value `GASPI_SUCCESS` *error_ message* contains the error message corresponding to the error code *error_ code.*

In case of error, the return value is `GASPI_ERROR`.

The procedure can be invoked in any of the Gaspi execution phases.

15 Profiling Interface

The profiling interface of GASPI consists of two parts. The statistics part provides the means to allow the user to collect basic profiling data about a program run.

15.1 Statistics

15.1.1 gaspi_statistic_counter_get

The `gaspi_statistic_counter_get` procedure is a *synchronous local blocking* procedure which retrieves a statistical counter from the local GASPI process.

GASPI provides a typed enumeration to determine a particular statistical counter.

```
typedef enum { GASPI_STATISTIC_BYTES_WRITTEN
             , GASPI_STATISTIC_BYTES_READ
             , GASPI_STATISTIC_BYTES_SENT
             , GASPI_STATISTIC_BYTES_RECEIVED
             } gaspi_statistic_counter_t;
```

A GASPI implementation is free to extend the above enumeration.

```
gaspi_return_t
gaspi_statistic_counter_get ( gaspi_statistic_counter_t counter
                            , gaspi_statistic_value_t value
                            )
```

Parameter:

(in) counter: the counter to be retrieved

(out) value: the current value of the counter

Execution phase:

Working

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_ERROR`: operation has finished with an error ⌟

After successful procedure completion, i. e. return value `GASPI_SUCCESS` *value* contains the current value of the corresponding counter.

15.1.2 `gaspi_statistic_counter_reset`

The `gaspi_statistic_counter_reset` procedure is a *synchronous local blocking* procedure which sets a statistical counter to 0.

```
gaspi_return_t
gaspi_statistic_counter_reset (gaspi_statistic_counter_t counter)
```

Parameter:

(in) counter: the counter to be reset

Execution phase:

Working

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_ERROR`: operation has finished with an error ⌟

15.1.3 `gaspi_statistic_counter_collect`

The `gaspi_statistic_counter_collect` procedure is a *synchronous collective time-based blocking* procedure.

```
gaspi_return_t
gaspi_statistic_counter_collect ( gaspi_statistic_counter_t counter
                                , gaspi_statistic_value_t value
                                , gaspi_group_t group
                                , gaspi_timeout_t timeout
                                )
```

Parameter:

(in) counter: the counter to be retrieved

(out) value: the current value of the counter

(in) group: the group of ranks which participate in the collect operation

(in) timeout: the timeout

Execution phase:

Working

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_TIMEOUT`: operation has run into timeout

GASPI_ERROR: operation has finished with an error ⌟

The behavior of gaspi_statistic_counter_collect is equivalent to a call to gaspi_allreduce (see 12.3.1) with num = 1, operation = GASPI_OP_SUM and datatype = GASPI_TYPE_ULONG.

After successful procedure completion, i. e. return value GASPI_SUCCESS, *value* contains the sum of the values of the corresponding *counter* of all ranks being a member of *group*. The counter values delivered by every participating rank are equivalent to the values which would be returned by local calls to gaspi_statistic_counter_get. That is, local calls to gaspi_statistic_counter_reset are respected.

Calls to gaspi_statistic_counter_collect do not contribute to the values of the collected counters at every rank.

In case of GASPI_TIMEOUT, the reduction operation is not finished yet, i. e. not all data is available that is needed to calculate the result. *value* has an undefined value.

In case of error, the return value is GASPI_ERROR. The error vector should be examined. *value* has an undefined value.

A Listings

A.1 success_or_die

Listing 11: success_or_die.h

```
#ifndef _SUCCESS_OR_DIE_H
#define _SUCCESS_OR_DIE_H 1

void success_or_die ( const char* file, const int line
                    , const int ec
                    );

#ifndef NDEBUG
#define ASSERT(ec) success_or_die (__FILE__, __LINE__, ec)
#else
#define ASSERT(ec) ec
#endif

#endif
```

Listing 12: success_or_die.c

```
#include <success_or_die.h>
#include <stdlib.h>
#include <stdio.h>
#include <GASPI.h>

```

```
void success_or_die ( const char* file, const int line
                    , const int ec
                    )
{
  if (ec != GASPI_SUCCESS)
    {
      gaspi_string_t str = GASPI_NOSTRING;

      gaspi_error_message (ec, &str);

      fprintf (stderr, "error in %s[%i]: %s\n", file, line, str);

      exit (EXIT_FAILURE);
    }
}
```

A.2 wait_if_queue_full

Listing 13: wait_if_queue_full.h

```
#ifndef _WAIT_IF_QUEUE_FULL_H
#define _WAIT_IF_QUEUE_FULL_H 1

#include <GASPI.h>

void wait_if_queue_full ( const gaspi_queue_id_t queue_id
                        , const gaspi_number_t request_size
                        );

#endif
```

Listing 14: wait_if_queue_full.c

```
#include <wait_if_queue_full.h>
#include <success_or_die.h>

void wait_if_queue_full ( const gaspi_queue_id_t queue_id
                        , const gaspi_number_t request_size
                        )
{
  gaspi_number_t queue_size_max = GASPI_NOQUEUESIZE;
  gaspi_number_t queue_size = GASPI_NOQUEUESIZE;

  ASSERT (gaspi_queue_size_max (queue_id, &queue_size_max));
  ASSERT (gaspi_queue_size (queue_id, &queue_size));

  if (queue_size + request_size >= queue_size_max)
    {
      ASSERT (gaspi_wait (queue_id, GASPI_BLOCK));
    }
}
```