```
MPI_Comm_rank(multiple_server_comm, &rank);
color = rank % num_servers;

/* Split the inter-communicator */
MPI_Comm_split(multiple_server_comm, color, rank,
               &single_server_comm);
```

The following is the corresponding server code:

```
/* Server code */
MPI_Comm  multiple_client_comm;
MPI_Comm  single_server_comm;
int       rank;

/* Create inter-communicator with clients and servers:
   multiple_client_comm */
...

/* Split the inter-communicator for a single server per group
   of clients */
MPI_Comm_rank(multiple_client_comm, &rank);
MPI_Comm_split(multiple_client_comm, rank, 0,
               &single_server_comm);
```

MPI_COMM_SPLIT_TYPE(comm, split_type, key, info, newcomm)

IN	comm	communicator (handle)
IN	split_type	type of processes to be grouped together (integer)
IN	key	control of rank assignment (integer)
INOUT	info	info argument (handle)
OUT	newcomm	new communicator (handle)

C binding

```
int MPI_Comm_split_type(MPI_Comm comm, int split_type, int key, MPI_Info info,
              MPI_Comm *newcomm)
```

Fortran 2008 binding

```
MPI_Comm_split_type(comm, split_type, key, info, newcomm, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, INTENT(IN) :: split_type, key
    TYPE(MPI_Info), INTENT(IN) :: info
    TYPE(MPI_Comm), INTENT(OUT) :: newcomm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_COMM_SPLIT_TYPE(COMM, SPLIT_TYPE, KEY, INFO, NEWCOMM, IERROR)
    INTEGER COMM, SPLIT_TYPE, KEY, INFO, NEWCOMM, IERROR
```

This function partitions the group associated with comm into disjoint subgroups such that each subgroup contains all MPI processes in the same grouping referred to by split_type.

Within each subgroup, the MPI processes are ranked in the order defined by the value of the argument key, with ties broken according to their rank in the old group. A new communicator is created for each subgroup and returned in newcomm. This is a collective call. All MPI processes in the group associated with comm must provide the same split_type, but each MPI process is permitted to provide different values for key. An exception to this rule is that an MPI process may supply the type value MPI_UNDEFINED, in which case MPI_COMM_NULL is returned in newcomm for such MPI process. No cached information propagates from comm to newcomm and no virtual topology information is added to the created communicators.

For split_type, the following values are defined by MPI:

MPI_COMM_TYPE_SHARED: all MPI processes in the group of newcomm are part of the same *shared memory domain* and can create a *shared memory segment* (e.g., with a successful call to MPI_WIN_ALLOCATE_SHARED). This segment can subsequently be used for load/store accesses by all MPI processes in newcomm.

> *Advice to users.* Since the location of some of the MPI processes may change during the application execution, the communicators created with the value MPI_COMM_TYPE_SHARED before this change may not reflect an actual ability to share memory between MPI processes after this change. (*End of advice to users.*)

MPI_COMM_TYPE_HW_GUIDED: this value specifies that the communicator comm is split according to a **hardware resource type** (for example a computing core or an L3 cache) specified by the "mpi_hw_resource_type" info key. Each output communicator newcomm corresponds to a single instance of the specified hardware resource type. The MPI processes in the group associated with the output communicator newcomm utilize that specific hardware resource type instance, and no other instance of the same hardware resource type.

If an MPI process does not meet the above criteria, then MPI_COMM_NULL is returned in newcomm for such MPI process.

MPI_COMM_NULL is also returned in newcomm in the following cases:

- MPI_INFO_NULL is provided.
- The info handle does not include the key "mpi_hw_resource_type".
- The MPI implementation neither recognizes nor supports the info key "mpi_hw_resource_type".
- The MPI implementation does not recognize the value associated with the info key "mpi_hw_resource_type".

The MPI implementation will return in the group of the output communicator newcomm the largest subset of MPI processes that match the splitting criterion.

The MPI processes in the group associated with newcomm are ranked in the order defined by the value of the argument key with ties broken according to their rank in the group associated with comm.

> *Advice to users.* The set of hardware resources that an MPI process is able to utilize may change during the application execution (e.g., because of the relocation of an MPI process), in which case the communicators created with the value

MPI_COMM_TYPE_HW_GUIDED before this change may not reflect the utilization of hardware resources of such MPI process at any time after the communicator creation. (*End of advice to users.*)

The user explicitly constrains with the info argument the splitting of the input communicator comm. To this end, the info key "mpi_hw_resource_type" is reserved and its associated value is an implementation-defined string designating the type of the requested hardware that follows the URI format described in Section 9.1.2 (e.g., "hwloc://NUMANode", "hwloc://Package" or "hwloc://L3Cache").

The value "mpi_shared_memory" is reserved and its use is equivalent to using MPI_COMM_TYPE_SHARED for the split_type parameter.

Rationale. The value "mpi_shared_memory" is defined in order to ensure consistency between the use of MPI_COMM_TYPE_SHARED and the use of MPI_COMM_TYPE_HW_GUIDED. (*End of rationale.*)

All MPI processes must provide the same value for the info key "mpi_hw_resource_type".

Example 7.3. Splitting MPI_COMM_WORLD into NUMANode subcommunicators.

```
MPI_Info info;
MPI_Comm hwcomm;
int      rank;

MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Info_create(&info);
MPI_Info_set(info, "mpi_hw_resource_type", "hwloc://NUMANode");
MPI_Comm_split_type(MPI_COMM_WORLD,
                    MPI_COMM_TYPE_HW_GUIDED,
                    rank, info, &hwcomm);
```

MPI_COMM_TYPE_RESOURCE_GUIDED: this value specifies that the communicator comm is split according to a **hardware resource type** (for example a computing core or an L3 cache) specified by the "mpi_hw_resource_type" info key or to a **logical resource type** (for example a process set name, see Section 11.3.2) specified by the "mpi_pset_name" info key.

Each output communicator newcomm corresponds to a single instance of the specified resource type. The MPI processes in the group associated with the output communicator newcomm utilize that specific resource type instance, and no other instance of the same resource type.

If an MPI process does not meet the above criteria, then MPI_COMM_NULL is returned in newcomm for such process.

MPI_COMM_NULL is also returned in newcomm in the following cases:

- MPI_INFO_NULL is provided.
- The info handle includes neither the key "mpi_hw_resource_type" nor the key "mpi_pset_name".
- The MPI implementation neither recognizes nor supports the info keys "mpi_hw_resource_type" and "mpi_pset_name".

- The MPI implementation does not recognize the value associated with the info key "mpi_hw_resource_type" or "mpi_pset_name".

The MPI implementation will return in the group of the output communicator newcomm the largest subset of MPI processes that match the splitting criterion.

Advice to users. The set of resources that an MPI process is able to utilize may change during the application execution (e.g., because of the relocation of an MPI process), in which case the communicators created with the value MPI_COMM_TYPE_RESOURCE_GUIDED before this change may not reflect the utilization of resources of such process at any time after the communicator creation. (*End of advice to users.*)

The user explicitly constrains with the info argument the splitting of the input communicator comm. To this end, the following info keys are reserved and their associated values are implementation-defined strings designating the type of the requested resource. Only one of these info keys can be used in info at a time in a call to MPI_COMM_SPLIT_TYPE; use of more than one info key is erroneous.

"mpi_hw_resource_type" is used to specify the type of a requested hardware resource (e.g., "hwloc://NUMANode", "hwloc://Package" or "hwloc://L3Cache"). The value "mpi_shared_memory" is reserved and its use is equivalent to using MPI_COMM_TYPE_SHARED for the split_type parameter.

Rationale. The value "mpi_shared_memory" is defined in order to ensure consistency between the use of MPI_COMM_TYPE_SHARED and the use of MPI_COMM_TYPE_RESOURCE_GUIDED. (*End of rationale.*)

All MPI processes in the group of the input communicator comm must provide the same info key to perform the splitting action. All MPI processes in the group of the input communicator comm must provide the same value for the info key "mpi_hw_resource_type".

"mpi_pset_name" is used to specify the type of a requested logical resource through the utilization of a process set name (e.g., "app://ocean" or "app://atmos"). This process set name must be valid in the session from which the input communicator comm is derived. If this input communicator is not derived from a session, then MPI_COMM_NULL is returned in newcomm.

All MPI processes that are both in the group of the input communicator comm and in the process set identified by the given process set name must provide the same info key to perform the splitting action. All MPI processes that are both in the group of the input communicator comm and in the process set identified by the given process set name must provide the same value for the info key "mpi_pset_name".

Example 7.4. Splitting MPI_COMM_WORLD into NUMANode subcommunicators.

```
MPI_Info info;
MPI_Comm hwcomm;
int      rank;

MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

```
    MPI_Info_create(&info);
    MPI_Info_set(info, "mpi_hw_resource_type", "hwloc://NUMANode");
    MPI_Comm_split_type(MPI_COMM_WORLD,
                        MPI_COMM_TYPE_RESOURCE_GUIDED,
                        rank, info, &hwcomm);
```

MPI_COMM_TYPE_HW_UNGUIDED: the group of MPI processes associated with newcomm must be a *strict* subset of the group associated with comm and each newcomm corresponds to a single instance of a **hardware resource type** (for example a computing core or an L3 cache).

All MPI processes in the group associated with comm that utilize that specific hardware resource type instance—and no other instance of the same hardware resource type—are included in the group of newcomm.

If a given MPI process cannot be a member of a communicator that forms such a strict subset, or does not meet the above criteria, then MPI_COMM_NULL is returned in newcomm for this process.

Advice to implementors. In a high-quality MPI implementation, the number of different new valid communicators newcomm produced by this splitting operation should be minimal unless the user provides a key/value pair that modifies this behavior. The sets of hardware resource types used for the splitting operation are implementation-dependent, but should reflect the hardware of the actual system on which the application is currently executing. (*End of advice to implementors.*)

Rationale. If the hardware resources are hierarchically organized, calling this routine several times using as its input communicator comm the output communicator newcomm of the previous call creates a sequence of newcomm communicators in each MPI process, which exposes a hierarchical view of the hardware platform, as shown in Example 7.5. This sequence of returned newcomm communicators may differ from the sets of hardware resource types, as shown in the second splitting operation in Figure 7.3. (*End of rationale.*)

Advice to users. Each output communicator newcomm can represent a different hardware resource type (see Figure 7.3 for an example). The set of hardware resources an MPI process utilizes may change during the application execution (e.g., because of MPI process relocation), in which case the communicators created with the value MPI_COMM_TYPE_HW_UNGUIDED before this change may not reflect the utilization of hardware resources for such MPI process at any time after the communicator creation. (*End of advice to users.*)

If a valid info handle is provided as an argument, the MPI implementation sets the info key "mpi_hw_resource_type" for each MPI process in the group associated with a returned newcomm communicator and the info key value is an implementation-defined string that indicates the hardware resource type represented by newcomm. The same hardware resource type must be set in all MPI processes in the group associated with newcomm.

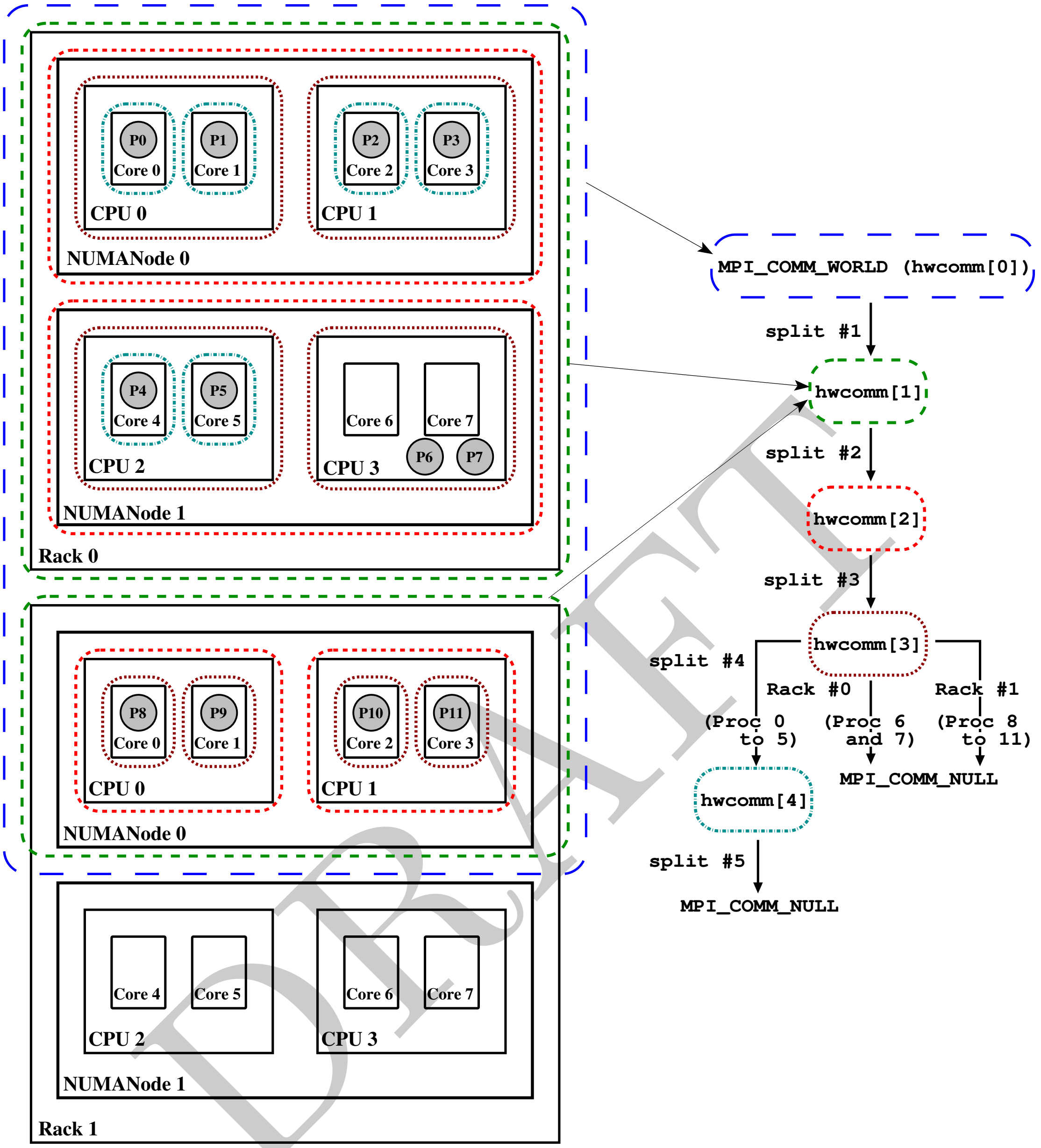

Figure 7.3: Recursive splitting of MPI_COMM_WORLD with MPI_COMM_SPLIT_TYPE and MPI_COMM_TYPE_HW_UNGUIDED. Dashed lines represent communicators whilst solid lines represent hardware resources. MPI processes (P0 to P11) utilize exclusively their respective core, except for P6 and P7, which utilize CPU #3 of Rack #0 and can therefore use Cores #6 and #7 indifferently. The second splitting operation yields two subcommunicators corresponding to NUMANodes in Rack #0 and to CPUs in Rack #1 because Rack #1 features only one NUMANode, which corresponds to the whole portion of the Rack that is included in MPI_COMM_WORLD and hwcomm[1]. For the first splitting operation, the hardware resource type returned in the info argument is "Rack" on the MPI processes on Rack #0, whereas on Rack #1, it can be either "Rack" or "NUMANode".

Example 7.5. Recursive splitting of MPI_COMM_WORLD.

```
#define MAX_NUM_LEVELS 32

MPI_Comm hwcomm[MAX_NUM_LEVELS];
int      rank, level_num = 0;

hwcomm[level_num] = MPI_COMM_WORLD;

while((hwcomm[level_num] != MPI_COMM_NULL)
      && (level_num < MAX_NUM_LEVELS-1)){
  MPI_Comm_rank(hwcomm[level_num],&rank);
  MPI_Comm_split_type(hwcomm[level_num],
                      MPI_COMM_TYPE_HW_UNGUIDED,
                      rank,
                      MPI_INFO_NULL,
                      &hwcomm[level_num+1]);
  level_num++;
}
```

Advice to implementors. Implementations can define their own split_type values, or use the info argument, to assist in creating communicators that help expose platform-specific information to the application. The concept of hardware-based communicators was first described by Träff [68] for SMP systems. Guided and unguided modes description as well as an implementation path are introduced by Goglin et al. [28]. (*End of advice to implementors.*)

MPI_COMM_CREATE_FROM_GROUP(group, stringtag, info, errhandler, newcomm)

IN	group	group (handle)
IN	stringtag	unique identifier for this operation (string)
IN	info	info object (handle)
IN	errhandler	error handler to be attached to new intra-communicator (handle)
OUT	newcomm	new communicator (handle)

C binding

```
int MPI_Comm_create_from_group(MPI_Group group, const char *stringtag,
              MPI_Info info, MPI_Errhandler errhandler, MPI_Comm *newcomm)
```

Fortran 2008 binding

```
MPI_Comm_create_from_group(group, stringtag, info, errhandler, newcomm, ierror)
    TYPE(MPI_Group), INTENT(IN) :: group
    CHARACTER(LEN=*), INTENT(IN) :: stringtag
    TYPE(MPI_Info), INTENT(IN) :: info
    TYPE(MPI_Errhandler), INTENT(IN) :: errhandler
    TYPE(MPI_Comm), INTENT(OUT) :: newcomm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Rationale. This function allows MPI implementations that do process migration to return the current processor. Note that nothing in MPI *requires* or defines process migration; this definition of MPI_GET_PROCESSOR_NAME simply allows such an implementation. (*End of rationale.*)

Advice to users. The user must provide at least MPI_MAX_PROCESSOR_NAME space to write the processor name—processor names can be this long. The user should examine the output argument, resultlen, to determine the actual length of the name. (*End of advice to users.*)

Inquire Hardware Resource Information

MPI_GET_HW_RESOURCE_INFO(hw_info)

OUT	hw_info	info object created (handle)

C binding

```
int MPI_Get_hw_resource_info(MPI_Info *hw_info)
```

Fortran 2008 binding

```
MPI_Get_hw_resource_info(hw_info, ierror)
    TYPE(MPI_Info), INTENT(OUT) :: hw_info
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_GET_HW_RESOURCE_INFO(HW_INFO, IERROR)
    INTEGER HW_INFO, IERROR
```

MPI_GET_HW_RESOURCE_INFO is a local procedure that returns an info object containing information pertaining to the hardware platform on which the calling MPI process is executing at the moment of the call. This information is stored as (key,value) pairs where each key is the name of a hardware resource type and its value is set to "true" if the calling MPI process is restricted to a single instance of a hardware resource of that type and "false" otherwise. The order in which the keys are stored in hw_info is unspecified. This procedure will return different information for MPI processes that are restricted to different hardware resources. Otherwise, info objects with identical (key, value) pairs are returned. The user is responsible for freeing hw_info via MPI_INFO_FREE.

Advice to users. The information returned in the info object might reflect the "hardware" resources presented to the application by a virtualization environment and may be restricted by access permissions or other constraints like environment variables and OS settings. (*End of advice to users.*)

The keys stored in the hw_info object have a *Uniform Resource Identifier* (URI) format. The first part of the URI indicates the key provider and the second part conforms to the format used by this key provider. The key provider "mpi://" is reserved for exclusive use by the MPI standard.

Advice to implementors. Key provider names could be derived from MPI implementation names (e.g., "mpich://", "openmpi://"), from names of external libraries or pieces of software (e.g., "hwloc://", "pmix://"), from names of programming or execution models (e.g., "openmp://"), from resource manager names (e.g., "slurm://") or from hardware vendor names. (*End of advice to implementors.*)

Advice to users. Users should be cautious when using such keys because comparisons between different providers may not be always meaningful or relevant. Also, the same hardware resource can be listed by multiple providers under different names.

One provider could convey types that represent individual hardware resource instances – for example, "provider_1://core/FF53C8A9" or "provider_1://numanode/2" – while another provider could provide types that represent categories or locations of hardware resources – for example, "provider_2://core" or "provider_2://numanode".

It is anticipated that, for MPI_COMM_SPLIT_TYPE, types that represent categories or locations will be more useful than types that represent individual resources. (*End of advice to users.*)

Advice to users. The keys stored in the info object returned by this procedure can be used in MPI_COMM_SPLIT_TYPE with the split_type value MPI_COMM_TYPE_HW_GUIDED or MPI_COMM_TYPE_RESOURCE_GUIDED as key *values* for the info key "mpi_hw_resource_type". (*End of advice to users.*)

Subsequent calls to MPI_GET_HW_RESOURCE_INFO may return different information throughout the execution of the program because an MPI process can be relocated (e.g., migrated or have its hardware restrictions changed).

Example 9.1. Splitting MPI_COMM_WORLD into NUMANode subcommunicators.

```
MPI_Info hw_info;
int      nb_keys   = 0, flag = 0;
int      is_found  = 0, is_restricted = 0;
int      valuelen  = 6; // max length between "false" and "true" + 1
char     *value    = calloc(valuelen, sizeof(char));
char     *hw_type  = calloc((MPI_MAX_INFO_KEY+1), sizeof(char));

MPI_Get_hw_resource_info(&hw_info);

MPI_Info_get_nkeys(hw_info, &nb_keys);
for(int index = 0 ; index < nb_keys ; index++){
  MPI_Info_get_nthkey(hw_info, index, hw_type);
  MPI_Info_get_string(hw_info, hw_type, &valuelen, value, &flag);
  if(strcmp(hw_type, "hwloc://NUMANode") == 0){
    is_found = 1;
    if(strcmp(value,"true") == 0)
      is_restricted = 1;
    break; // Resource of type NUMANode found
  }
}

// The calling MPI process is restricted to a resource
// of the chosen type (NUMANode)
```

```
if(is_found  && is_restricted){
   MPI_Comm hw_comm;
   MPI_Info split_info;
   int rank;

   MPI_Info_create(&split_info);
   MPI_Info_set(split_info, "mpi_hw_resource_type", hw_type);
   // hw_type now serves as value for the "mpi_hw_resource_type" key
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_split_type(MPI_COMM_WORLD, MPI_COMM_TYPE_RESOURCE_GUIDED,
                              rank, split_info, &hw_comm);

  // Use hw_comm from this point
}
```

9.2 Memory Allocation

In some systems, message-passing and remote-memory-access (RMA) operations run faster when accessing specially allocated memory (e.g., memory that is shared by the other processes in the communicating group on an SMP). MPI provides a mechanism for allocating and freeing such special memory. The use of such memory for message-passing or RMA is not mandatory, and this memory can be used without restrictions as any other dynamically allocated memory. However, implementations may restrict the use of some RMA functionality as defined in Section 12.5.3.

MPI_ALLOC_MEM(size, info, baseptr)

IN	size	size of memory segment in bytes (non-negative integer)
IN	info	info argument (handle)
OUT	baseptr	pointer to beginning of memory segment allocated

C binding

```
int MPI_Alloc_mem(MPI_Aint size, MPI_Info info, void *baseptr)
```

Fortran 2008 binding

```
MPI_Alloc_mem(size, info, baseptr, ierror)
    USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size
    TYPE(MPI_Info), INTENT(IN) :: info
    TYPE(C_PTR), INTENT(OUT) :: baseptr
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_ALLOC_MEM(SIZE, INFO, BASEPTR, IERROR)
    INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
    INTEGER INFO, IERROR
```