`ryo_arg_free`: the routine to free the *C* structure created by the argument decode routine for the operation;

`ryo_result`: non-zero if this operation returns a result;

`ryo_res_encode/ryo_res_decode`: the address of the routines to encode or decode (respectively) the operation's result from a *C* structure to its corresponding abstract syntax (present only if the operation returns a result);

`ryo_res_free`: the routine to free the *C* structure created by the result decode routine for the operation; and,

`ryo_errors`: a pointer to null-terminated list of the errors which this operation may generate (present only if the operation may returns at least one error).

The `RyError` structure is used to define the compiled parts of an error:

```
struct RyError {
    char   *rye_name;
    int     rye_err;

    IFP     rye_param_encode;
    IFP     rye_param_decode;
    IFP     rye_param_free;
};
```

The elements of this structure are:

`rye_name/rye_err`: the error name and code;

`rye_param_encode/rye_res_decode`: the address of the routines to encode or decode (respectively) the error's parameter from a *C* structure to its corresponding abstract syntax (present only if the error contains a parameter); and,

`ryo_param_free`: the routine to free the *C* structure created by the parameter decode routine for the error.

8.3 Routines for Initiators

Initiators use the routines in the *libacsap*(3n) library for association control. These are described in Chapter 2 of *Volume One*.

8.4 Routines for Invokers

After forming an association, invokers are able to request operations for execution by the performer. Two interfaces are provided: a synchronous interface and an asynchronous interface.

Synchronous Interface

Most invokers desire a simple interface for synchronous execution of operations (the operation blocks until it is completed). The `RyOperation` routine is used to provide such an interface.

```
int     RyOperation (sd, ryo, op, in, out, response, roi)
int     sd;
struct RyOperation *ryo;
int     op,
       *response;
caddr_t in,
       *out;
struct RoSAPindication *roi;
```

The parameters to this procedure are:

`sd:` the association-descriptor;

`ryo:` a pointer to a `RyOperation` table;

`op:` the operation code of the operation to invoke;

`response:` an integer-valued location which is updated if the call succeeds;

`in:` the address of the *C* structure for the operation's argument;

`out`: an address-valued location which is updated if the call succeeds; and,

`roi`: a pointer to a `RoSAPindication` structure, which is updated only if the operation was not performed.

The `RyOperation` routine returns one of three values: `NOTOK` (on failure), `OK` (on reading a reply), or `DONE` (when something else happens).

If the call to `RyOperation` returns the manifest constant `NOTOK`, then the `RoSAPpreject` structure contained in the `RoSAPindication` parameter `roi` contains the reason for the failure.

If the call to `RyOperation` returns the manifest constant `OK`, then the `response` and `out` parameters are updated. If the value of the integer pointed to by `response` is `RY_RESULT`, then the value of the address pointed to by `out` is a newly allocated *C* structure for the operation's result (if any). If the value of the integer pointed to by `response` is any other value, then it indicates the error which the operation returned, and the value of the address pointed to by `out` is a newly allocated *C* structure for the error's parameter (if any). Note that if a result or parameter is returned, then it is the invoker's responsibility to release the memory used by the *C* structure, when appropriate.

If the call to `RyOperation` returns the manifest constant `DONE`, then an indication that the association is pending termination is returned. The value of the `roi_type` element of the `RoSAPindication` parameter is either `ROI_END` or `ROI_FINISH`. Consult Section 3.3.4 of *Volume One* for the details, as this normally does not happen to initiators.

The routine `RyOperation` is simply a comprehensive interface for a more general routine, `RyOpInvoke`, which invokes an operation:

```
int     RyOpInvoke (sd, ryo, op, in, out, rfx, efx, class,
                    invokeID, linkedID, priority, roi)
int     sd;
struct RyOperation *ryo;
int     op,
        class,
        invokeID,
       *linkedID,
        priority;
```

```
IFP     rfx,
        efx;
caddr_t in,
       *out;
struct RoSAPindication *roi;
```

The parameters to this procedure are:

`sd:` the association-descriptor;

`ryo:` a pointer to a `RyOperation` table;

`op:` the operation code of the operation to invoke;

`in:` the address of the *C* structure for the operation's argument;

`out:` an address-valued location which is updated if the call succeeds;

`rfx:` the address of a dispatch routine to be invoked if the operation returns a result;

`efx:` the address of a dispatch routine to be invoked if the operation returns an error;

`class:` the operation class (either `ROS_SYNC` for a synchronous operation, or `ROS_ASYNC` for an asynchronous one);

`invokeID:` the invocation ID of this request;

`linkedID:` the linked ID of this request (use `NULLIP` if not linked);

`priority:` the priority of this request (use `ROS_NOPRIO` if the priority is undetermined); and,

`roi:` a pointer to a `RoSAPindication` structure, which is updated only if the operation was not performed.

This routine is exactly equivalent to the `RoInvokeRequest` routine in the *librosap*(3n) library, with the additional functions of: automatically encoding and decoding arguments, results, and parameters. Hence, the routine `RyOperation`, is essentially a call to `RyOpInvoke` with a few hard-wired parameters:

`class:` `ROS_SYNC`;

`invokeID:` a unique-number;

`linkedID:` `NULLIP`; and,

`priority:` `ROS_NOPRIO`.

Asynchronous Interface

Although most invokers desire a simple interface for synchronous execution, it is also useful to have an asynchronous interface. Further, since a synchronous interface can be thought of as a degenerate subset of an asynchronous interface, most invokers use the asynchronous interface and simply ask for it to block after making the invocation. The `RyStub` routine is used to provide such an interface.

```
int     RyStub (sd, ryo, op, id, linked, in, rfx, efx,
                class, roi)
int     sd;
struct RyOperation *ryo;
int     op,
        id,
       *linked,
        class;
caddr_t in;
IFP     rfx,
        efx;
struct RoSAPindication *roi;
```

The parameters to this procedure are:

`sd:` the association-descriptor;

`ryo:` a pointer to a `RyOperation` table;

`op:` the operation code of the operation to invoke;

`id:` the invocation identifier to use;

`linked:` the linked invocation identifier to use, if any (use `NULLIP` if this is not a linked reply);

`in:` the address of the *C* structure for the operation's argument;

`rfx:` the address of a dispatch routine to be invoked if the operation returns a result;

`efx:` the address of a dispatch routine to be invoked if the operation returns an error or is rejected;

`class:` the operation class (either `ROS_SYNC` for a synchronous operation, `ROS_ASYNC` for an asynchronous operation, or `ROS_INTR` for a synchronous but user-interruptable operation); and,

`roi:` a pointer to a `RoSAPindication` structure, which is updated only if the operation was not performed.

The `RyStub` routine returns one of three values: `NOTOK` (on failure), `OK` (on success), or `DONE` (when something else happens).

If the call to `RyStub` returns the manifest constant `NOTOK`, then the `RoSAPpreject` structure contained in the `RoSAPindication` parameter `roi` contains the reason for the failure.

If the call to `RyStub` returns the manifest constant `OK`, then the operation was queued or completed, depending on the value of the `class` parameter. If the value of the `class` parameter was `ROS_ASYNC`, then `RyStub` returns immediately after queuing the invocation. Otherwise, `RyStub` waits for a response to the operation, invokes the appropriate dispatch routine, and then returns. Both the result and error dispatch routines are invoked in the same way:

```
result = (*fnx) (sd, id, reason, value, roi)
int     sd;
int     id,
        reason;
caddr_t value;
struct RoSAPindication *roi;
```

The parameters to this procedure are:

`sd:` the association-descriptor;

`id:` the invocation identifier associated with the result or error;

`reason:` either `RY_RESULT` if the result dispatch routine is invoked, the error code if the error dispatch routine is invoked with an error, or `RY_REJECT` if the error dispatch routine is invoked with a rejection;

`value:` the address of the *C* structure for the result or error parameter (if any); and,

`roi:` a pointer to a `RoSAPindication` structure, which is updated only if the result, error, or rejection is faulty.

The dispatch routine should return the manifest constant `OK` unless you know what you're doing. Note that if `value` is present, then the *librosy*(3n) library will automatically release the memory used by the *C* structure when the dispatch routine returns.

If the call to `RyStub` returns the manifest constant `DONE`, then an indication that the association is pending termination is returned. The value of the `roi_type` element of the `RoSAPindication` parameter is either `ROI_END` or `ROI_FINISH`. Consult Section 3.3.4 of *Volume One* for the details, as this normally does not happen to initiators.

If the value of the `class` parameter was `ROS_INTR`, then `RyStub` issues a synchronous call (as described above for the `ROS_SYNC` case). However, if the user generates an interrupt, usually by typing control-C (`^C`), while waiting for the response to the operation, then `RyStub` will return immediately with the value `NOTOK` and the `rop_reason` element of the `RoSAPpreject` structure contained in the `RoSAPindication` parameter `roi` will be set to the value `ROS_INTERRUPTED`.

At this time, the program may do other processing and later return to wait for the outcome of the operation. Instead, the program may direct the run-time environment to silently discard any outcome. This is useful when the user wishes to abandon an inprogress operation and really doesn't care what the results are (i.e., when reading from a database). The verb"RyDiscard" routine is used to accomplish this function:

```
int     RyDiscard (sd, id, roi)
int     sd,
```

```
            id;
    struct RoSAPindication *roi;
```

The parameters to this procedure are:

`sd:` the association descriptor;

`sd:` the invocation-identifier associated with the operation; and,

`roi:` a pointer to a `RoSAPindication` structure, which is updated only on failure.

The `RyDiscard` routine returns either `NOTOK` (on failure) or `OK` (on success).

8.5 Routines for Responders

Responders use the routines in the *libacsap*(3n) library for association control. These are described in Chapter 2 of *Volume One*.

8.6 Routines for Performers

After forming an association, performers register the operations they are willing to perform. They then typically wait for those operations to be invoked.

8.6.1 Registering Operations

The routine `RyDispatch` is used to register a handler for an operation:

```
int     RyDispatch (sd, ryo, op, fnx, roi)
int     sd;
struct RyOperation *ryo;
int     op;
IFP     fnx;
struct RoSAPindication *roi;
```

The parameters to this procedure are:

`sd:` the association-descriptor;

`ryo`: a pointer to a `RyOperation` table;

`op`: the operation code of the operation to be registered;

`fnx`: the address of a dispatch routine to be invoked when the operation is invoked; and,

`roi`: a pointer to a `RoSAPindication` structure, which is updated only if the call fails.

An operation may be un-registered by using `NULLIFP` for the `fnx` parameter.

When a registered operation is invoked (from the routine `RyWait` described in Section 8.7), the dispatch routine is invoked with five parameters:

```
result = (*fnx) (sd, ryo, rox, in, roi)
int     sd;
struct RyOperation *ryo;
struct RoSAPinvoke *rox;
caddr_t in;
struct RoSAPindication *roi;
```

The parameters to this procedure are:

`sd`: the association-descriptor;

`ryo`: a pointer to a entry in the `RyOperation` table for this operation;

`rox`: a pointer to a `RoSAPinvoke` structure containing miscellaneous information regarding the operation being invoked (e.g., the invocation id);

`in`: the address of the *C* structure for the operation's argument (if any); and,

`roi`: a pointer to a `RoSAPindication` structure, which is updated only if the operation was not performed.

Note that if an argument is present for the operation, then the *librosy*(3n) library will automatically release the memory used by the *C* structure when the dispatch routine returns.

The return value of the dispatch routine is consulted to determine the disposition of the invocation. One of three values should be returned: `NOTOK`, which indicates that some non-operational error occured; `OK`, which indicates that the operation was handled; or, `DONE`, which indicates that the association is pending termination.

8.6.2 Responding to Operations

If the operation corresponding to the dispatch routine does not return a result, then the routine may simply process the request and return. Otherwise, the dispatch routine should ultimately cause one of three routines to be called to handle an invocation.

The routine `RyDsResult` is used to return a result to an invocation.

```
int     RyDsResult (sd, id, out, priority, roi)
int     sd;
int     id,
        priority;
caddr_t out;
struct RoSAPindication *roi;
```

The parameters to this procedure are:

`sd:` the association-descriptor;

`id:` the ID of the invocation being responded to;

`out:` the address of the *C* structure for the operation's result (if any);

`priority:` the priority of the response (use `ROS_NOPRIO` if the priority is undetermined); and,

`roi:` a pointer to a `RoSAPindication` structure, which is updated only if the call fails.

The routine `RyDsError` is used to return an error to an invocation.

```
int     RyDsError (sd, id, err, out, priority, roi)
int     sd;
int     id,
```