```
int     RtWaitRequest (sd, secs, rti)
int     sd;
int     secs;
struct RtSAPindication *rti;
```

The parameters to this procedure are:

`sd:` the association-descriptor;

`secs:` the maximum number of seconds to wait for the event (a value of `NOTOK` indicates that the call should block indefinitely, whereas a value of `OK` indicates that the call should not block at all, e.g., a polling action); and,

`rti:` a pointer to a `RtSAPindication` structure, which is always updated.

Unlike the other routines in the *librtsap*(3n) library, the `RtWaitRequest` routine returns one of three values: `NOTOK` (on failure), `OK` (on reading data) or `DONE` (when something else happens).

If the call to `RtWaitRequest` returns the manifest constant `NOTOK`, then the `RtSAPabort` structure contained in the `RtSAPindication` parameter `rti` contains the reason for the failure.

Otherwise if the call to `RoWaitRequest` returns the manifest constant `OK`, then data has arrived. This event is encoded in the `rti` parameter, depending on the value of the `rti_type` element. Currently, when `RoWaitRequest` returns `OK`, the `rti_type` element is set to `RTI_TRANSFER`, which indicates that a RT-TRANSFER.REQUEST event has occurred.

Otherwise if the call to `RtWaitRequest` returns the manifest constant `DONE`, then some event other than data arriving has occurred. This event is also encoded in the `rti` parameter, depending on the value of the `rti_type` element. Currently, when `RoWaitRequest` returns `DONE`, the `rti_type` element is set to one of two values:

Value	Event
`RTI_TURN`	RT-TURN-PLEASE.INDICATION
`RTI_TURN`	RT-TURN-GIVE.INDICATION
`RTI_CLOSE`	X.410 CLOSE.INDICATION (for old-style associations)
`RTI_FINISH`	RT-CLOSE.INDICATION

Transfer Indication

When an RT-TRANSFER.INDICATION event occurs, the `rti_type` field of the `rti` parameter contains the value `RTI_TRANSFER`, and a `RtSAPtransfer` structure is contained inside the `rti` parameter.

```
struct RtSAPtransfer {
    PE      rtt_data;
};
```

The elements of this structure are:

`rtt_data`: the data received.

Note that the `rtt_data` element is allocated via *malloc*(3) and should be released using the `RTTFREE` macro when no longer referenced. The `RTTFREE` macro behaves as if it was defined as:

```
void    RTTFREE (rtt)
struct RtSAPtransfer *rtt;
```

The macro frees only the data allocated in the `RtSAPtransfer` structure and not the structure itself.

Turn Indication

When an RT-TURN-GIVE.INDICATION or RT-TURN-PLEASE.INDICATION events occur, the `rti_type` field of the `rti` parameter contains the value `RTI_TURN`, and a `RtSAPturn` structure is contained inside the `rti` parameter.

```
struct RtSAPturn {
    int     rtu_please;

    int     rtu_priority;
};
```

The elements of this structure are:

`rtu_please`: if set, indicates that a RT-PLEASE-TURN.INDICATION event has occurred; if not set, indicates that a RT-GIVE-TURN.INDICATION event has occurred; and,

`rtu_priority`: the priority at which the turn is requested (meaningful only if `rtu_please` is set).

Close Indication

When an X.410 CLOSE.INDICATION event occurs, the `rti_type` field of the `rti` parameter contains the value `RTI_CLOSE`, and a `RtSAPclose` structure is contained inside the `rti` parameter.

```
struct RtSAPclose {
    int     rtc_dummy;
};
```

Finish Indication

When an RT-CLOSE.INDICATION event occurs, the `rti_type` field of the `rti` parameter contains the value `RTI_FINISH`, and a `AcSAPfinish` structure is contained inside the `rti` parameter.

4.2.3 Managing the Turn

The user with the turn is permitted to send data to the other user. To request the turn, one invokes the `RtPTurnRequest` routine, which corresponds to the RT-PLEASE-TURN.REQUEST.

```
int     RtPTurnRequest (sd, priority, rti)
int     sd;
int     priority;
struct RtSAPindication *rti;
```

The parameters to this procedure are:

`sd:` the association-descriptor;

`priority:` the priority at which the turn is requested (this is defined by each application); and,

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

If the call to the `RtPTurnRequest` routine succeeds, then the turn has been requested of the remote user. Otherwise the `RtSAPabort` structure contained in the `RtSAPindication` parameter `rti` contains the reason for failure.

To relinquish the turn, one invokes the `RtGTurnRequest` routine, which corresponds to the RT-GIVE-TURN.REQUEST.

```
int     RtGTurnRequest (sd, rti)
int     sd;
struct RtSAPindication *rti;
```

The parameters to this procedure are:

`sd:` the association-descriptor; and,

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

If the call to the `RtGTurnRequest` routine succeeds, then the turn has been relinquished to the remote user. Otherwise the `RtSAPabort` structure contained in the `RtSAPindication` parameter `rti` contains the reason for failure.

4.2.4 Asynchronous Event Handling

The events discussed thus far have been synchronous in nature. Some users of the reliable transfer service may wish an asynchronous interface. The `RtSetIndications` routine is used to change the service associated with an association-descriptor to an asynchronous interface.

```
int   RtSetIndications (sd, indication, rti)
int   sd;
IFP   indication;
struct RtSAPindication *rti;
```

The parameters to this procedure are:

`sd:` the association-descriptor;

`indication:` the address of an event-handler routine to be invoked when an event occurs; and,

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

If the service is to be made asynchronous, then `indication` is specified; otherwise, if the service is to be made synchronous, it is not specified (use the manifest constant `NULLIFP`). The most likely reason for the call failing is `RTS_WAITING`, which indicates that an event is waiting for the user.

When an event occurs, the event-handler routine is invoked with two parameters:

```
(*handler) (sd, rti);
int     sd;
struct RtSAPindication *rti;
```

The parameters are:

`sd:` the association-descriptor; and,

`rti:` a pointer to a `RtSAPindication` structure encoding the event.

Note that the data contained in the structure was allocated via *malloc*(3), and should be released with the appropriate macro (either `RTTFREE` or `RTPFREE`) when no longer needed.

When an event-handler is invoked, future invocations of the event-hander are blocked until it returns. The return value of the event-handler is ignored. Further, during the execution of a synchronous call to the library, the event-handler will be blocked from being invoked.

> **NOTE:** The *librtsap*(3n) library uses the SIGEMT signal to provide these services. Programs using asynchronous association-descriptors should NOT use SIGEMT for other purposes.

4.2.5 Synchronous Event Multiplexing

A user of the reliable transfer service may wish to manage multiple association-descriptors simultaneously; the routine `RtSelectMask` is provided for this purpose. This routine updates a file-descriptor mask and associated counter for use with `xselect` (consult Section 2.4), as assocation-descriptors are file-descriptors.

```
int     RtSelectMask (sd, mask, nfds, rti)
int     sd;
fd_set *mask,
int    *nfds;
struct RtSAPindication *rti;
```

The parameters to this procedure are:

`sd:` the association-descriptor;

`mask`: a pointer to a file-descriptor mask meaningful to `xselect`;

`nfds`: a pointer to an integer-valued location meaningful to `xselect`; and,

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

If the call is successful, then the `mask` and `nfds` parameters can be used as arguments to `xselect`. The most likely reason for the call failing is `RTS_WAITING`, which indicates that an event is waiting for the user.

If `xselect` indicates that the association-descriptor is ready for reading, `RtWaitRequest` should be called with the `secs` parameter equal to `OK`. If the network activity does not constitute an entire event for the user, then `RtWaitRequest` will return `NOTOK` with error code `RTS_TIMER`.

4.2.6 Reliable Transfer (revisited)

The mechanism provided by `RtTransferRequest` may not be useful for applications which have large amounts of data to transfer. In most cases, it is preferable to transfer data incrementally. To provide for this functionality, the routine `RtSetDownTrans` is provided:

```
int     RtSetDownTrans (sd, fnx, rti)
int     sd;
IFP     fnx;
struct RtSAPindication *rti;
```

The parameters to this procedure are:

`sd`: the association descriptor;

`fnx`: the address of an event-handler routine to be invoked when data is needed; and,

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

If the `fnx` parameter is some value other than the manifest constant `NULLIFP`, then upon successful return from `RtSetDownTrans`, the behavior of the routine `RtTransferRequest` is altered.

1. The `data` parameter to `RtTransferRequest` may be `NULLPE`.

2. In this case, the event-handler routine `fnx` will be invoked in order to retrieve a portion of the data to be transferred:

```
result = (*fnx) (sd, base, len, size, ack, ssn, rti);
int     sd;
char  **base;
int    *len,
        size;
long    ack,
        ssn;
struct RtSAPindication *rti;
```

   where `sd` is the association-descriptor which was given to the routine `RtTransferRequest`.

3. If the `base` parameter has the value `NULLVP`, then a RT-PLEASE.INDICATION event is being signaled, and the `size` parameter has the value of the priority associated with the event.

4. Otherwise, the `base` and `len` parameters point to a pointer/length pair which should be set by the event handler to upto `size` octets. The event handler is responsible for any memory allocation (e.g., allocating a buffer of `size` octets and then assigning the address of the buffer to `*base`). Once a buffer is chosen, the event handler should read upto `size` octets into the buffer and set `*len` to the number of octets actually read. The `ssn` and `ack` parameters given the values of the last synchronization point requested and acknolwedged (how this information should be used is unknown at this time). If the value assigned to `*len` is zero, then this indicates that all data has been read and the transfer should be completed.

5. If the value of the `size` is zero, then this indicates that the provider could not negotiate a incremental transfer and the event handler should allocate a buffer of arbitrary size, read all of the data to be transferred into that one buffer, and then update `*base` and `*len` accordingly.

6. If the event handler encounters an error, it should return the manifest constant `NOTOK` (otherwise, it should return `OK`). If an error is signaled,

the event handler should update the `rti` structure accordingly. The easiest way to do this is:

```
return rtsaplose (rti, RTS_TRANSFER, NULLCP, "text");
```

If the event is RT-PLEASE.REQUEST, then signaling an error results in the provider generating an S-ACTIVITY-INTERRUPT.REQUEST; otherwise when an error is signaled, the provider will generate an S-ACTIVITY-DISCARD.REQUEST to abort the transfer.

For similar reasons, the mechanism employed by `RtWaitRequest` may not be useful for applications which have large amounts of data to transfer. Again, in most cases it is preferable to transfer data incrementally. To provide for this functionality, the routine `RtSetUpTrans` is provided:

```
int     RtSetUpTrans (sd, fnx, rti)
int     sd;
IFP     fnx;
struct RtSAPindication *rti;
```

The parameters to this procedure are:

`sd:` the association descriptor;

`fnx:` the address of an event-handler routine to be invoked when data has been received; and,

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

If the `fnx` parameter is some value other than the manifest constant `NULLIFP`, then upon successful return from `RtSetUpTrans`, the behavior of the routine `RtWaitRequest` is altered when it returns an RT-TRANSFER.INDICATION (the `rti_type` field of the `rti` parameter contains the value `RTI_TRANSFER`, and a `RtSAPtransfer` structure is contained inside the `rti` parameter). No data is returned by `RtWaitRequest`, rather as data is received, the event routine is invoked:

1. When data is received, the event-handler routine `fnx` will be invoked in order to store a portion of the data being transferred:

```
result = (*fnx) (sd, event, addr, rti);
int     sd;
int     event;
caddr_t addr;
struct RtSAPindication *rti;
```

where `sd` is the association-descriptor which was given to the routine `RtWaitRequest`.

2. If the `event` parameter has the value `SI_DATA`, then the `addr` parameter is really a pointer to a `struct qbuf` structure, and the event handler should traverse the qbuf writing out the data found therein.

3. If the `event` parameter has the value `SI_SYNC`, then the `addr` parameter is really a pointer to a `struct SSAPsync` structure, and the event handler should note the information contained therein.Currently, this will only occur when a S-MINOR-SYNC.INDICATION event is signaled.

4. If the `event` parameter has the value `SI_ACTIVITY`, then the `addr` parameter is really a pointer to a `struct SSAPactivity` structure, and the event handler should note the information contained therein. Currently, there are four events that are signaled:

   - S-ACTIVITY-START.INDICATION which indicates that a transfer is about to begin;
   - S-ACTIVITY-END.INDICATION which indicates that a transfer is about to complete; and,
   - S-ACTIVITY-INTERRUPT.INDICATION and S-ACTIVITY-DISCARD.INDICATION which indicate that an activity is either suspended or aborted.

5. If the `event` parameter has the value `SI_REPORT`, then the `addr` parameter is really a pointer to a `struct SSAPreport` structure, and the event handler should note the information contained therein. Currently, this will only occur when a S-U-EXCEPTION-REPORT.INDICATION event is signaled.

6. If the event handler encounters an error, it should return the manifest constant `NOTOK` (otherwise, it should return `OK`). If an error is signaled,

the event handler should update the `rti` structure accordingly. The easiest way to do this is:

```
return rtsaplose (rti, RTS_TRANSFER, NULLCP, "text");
```

If the event is S-ACTIVITY-INTERRUPT.INDICATION, S-ACTIVITY-INTERRUPT.INDICATION, or S-ACTIVITY-INTERRUPT.INDICATION, then signaling an error is ignorred by the provider; otherwise, when an error is signaled, the provider will generate an S-U-EXCEPTION-REPORT.REQUEST to abort the transfer.

4.3 Error Conventions

All of the routines in this library return the manifest constant `NOTOK` on error, and also update the `rti` parameter given to the routine. The `rti_abort` element of the `RtSAPindication` structure encodes the reason for the failure. One coerces a pointer to a `RtSAPabort` structure, and consults the `rta_reason` element of this latter structure. This element can be given as a parameter to the routine `RtErrString` which returns a null-terminated diagnostic string.

```
char   *RtErrString (c)
int     c;
```

4.4 Compiling and Loading

Programs using the *librtsap*(3n) library should include `<isode/rtsap.h>`, which automatically includes the header file `<isode/psap.h>` described in Chapter 5. The programs should also be loaded with `-lisode`.

4.5 An Example

Let's consider how one might construct a generic server that uses reliable transfer services to establish an association, but then uses remote operation services to communicate with its peer. This entity will use a synchronous interface.