teklib.doc

COLLABORATORS			
	TITLE : teklib.doc		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		July 31, 2024	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

teklib.doc

1.1 teklib.doc

```
TAckMsg
TAddHead
TAddSockPort
TAddTag
TAddTail
TAllocSignal
TCreatePool
TCreatePort
TCreateTask
TDestroy
TDropMsg
TFindSockPort
TFirstNode
TFreeMsg
TFreeSignal
TGetMsg
TGetMsgAttrs
TGetMsgSize
TGetMsgStatus
TGetRandom
TGetRandomSeed
TGetTagArray
TGetTagValue
TInitList
TInitLock
TInitMMU
TInitMemHead
TInitTags
TInsert
TLastNode
TListEmpty
TLock
TMMUAlloc
TMMUAlloc0
TMMUAllocHandle
TMMUAllocHandle0
TMMUFree
TMMUFreeHandle
```

```
rendered with docco[tm] TEKlib document generator (simple amigaguide)
```

1.2 TCreateTask

```
    NAME
        TCreateTask - create task.

    SYNOPSIS
        task = TCreateTask(parenttask, function,  taglist);
        TAPTR              TAPTR       TTASKFUNC* TTAGITEM*
```

```
FUNCTION
    launch a task at the given function, or create an
    application's basetask.

    for creating a basetask, both parenttask and function must
    be TNULL. all further tasks and most TEKlib internal
    structures will be derived from a basetask in the end, so
    it's usually one of the first objects created in a TEKlib
    framework.

    for creating a child task, parenttask must refer to the
    caller's context, and function usually refers to a task
    entry function. sometimes it may be desirable to only call
    an init function in a new context, so function may be TNULL,
    provided that the tag argument TTask_InitFunc is specified.
    if neither a task entry function nor an init function is
    specified, TCreateTask returns TNULL.

INPUTS
    parenttask - parent task. for creating a child task, this must
                 refer to the current context. TNULL for creating
                 an application's base task.

    function   - function entry. this must refer to a function
                 with the prototype TVOID (*function)(TAPTR task),
                 or may optionally be TNULL when the tag
                 TTask_InitFunc is specified in the taglist
                 arguments. TNULL when a basetask is to be created.

    taglist    - pointer to an array of tag items.

TAGS
    TTask_UserData, TAPTR
        pointer to arbitrary user data. a task's userdata field
        can be obtained with TTaskGetData.
        default: TNULL

    TTask_InitFunc, TBOOL (*function)(TAPTR task)
        pointer to a user init function. TCreateTask will initially
        call this function inside a newly created context, and enter
        the task's main function entry only if the init function
        returns TTRUE. otherwise child task creation is entirely
        abandoned, the task entry function is never called, and
        TCreateTask returns TNULL. when this argument is specified,
        the task's function entry argument may be TNULL.
        default: TNULL

    TTask_CreatePort, TBOOL
        create an initial message-port in the child's context.
        TCreateTask will entirely fail and return TNULL when
        a childport was requested and could not be established.
        by default, any newly created task will be supplied with
        a messageport. default: TTRUE

    TTask_MMU, TAPTR
        pointer to a memory management unit for allocating the task's
```

```
            structures. default: the heap MMU of an application's
            basetask, or TNULL if this is the basetask to be created.

        TTask_HeapMMU, TAPTR
            pointer to a memory management unit. the new task's heap
            memory manager will be put on top of this MMU.
            default: the argument or default value to TTask_MMU

    RESULTS
        task       - task handle, or TNULL if the task could not be
                     established. a task handle is destroyed with a
                     call to TDestroy.

    SEE ALSO
        TDestroy, TTaskGetData, TTaskSetData, TTaskPort,
        TTaskBaseTask, TCreatePort, TTaskHeapMMU
```

1.3 TAllocSignal

```
    NAME
        TAllocSignal - allocate a single or a set of signals

    SYNOPSIS
        signals = TAllocSignal(task, prefsignals)
        TUINT                  TAPTR TUINT

    FUNCTION
        allocate a signal (or a set of preferred signals) from the
        given task. if prefsignals is 0, then this function will try to
        reserve any single free signal. if prefsignals is not 0, this
        function tries to reserve the exact set specified, and returns
        0 if any of the specified signals are already in use.

    INPUTS
        task        - task to which the signal (or signal set) will belong
        prefsignals - preferred signals to allocate, or zero

    RESULTS
        signals     - allocated signal mask. zero if out of signals, or
                      when any of prefsignals are already in use.

    NOTES
        signals no longer needed should be freed with TFreeSignal.

    SEE ALSO
        TFreeSignal, TSignal, TSetSignal, TWait
```

1.4 TFreeSignal

```
NAME
    TTaskFreeSignal - free a single or set of task signals

SYNOPSIS
    TFreeSignal(task, sigmask)
                TAPTR TUINT

FUNCTION
    free a single or set of signals and return it to a task's
    pool of allocatable signals.

INPUTS
    task    - task to which the signal(s) belong.
    sigmask - signal mask to be freed. it is safe to pass
              0 (no-signal) here.

SEE ALSO
    TAllocSignal
```

1.5 TSignal

```
NAME
    TSignal - submit a set of signals to a task.

SYNOPSIS
    TSignal(task, signals);
            TAPTR TUINT

FUNCTION
    submit signals to a task. when the task was waiting
    for the specified signals, it will resume operation.

INPUTS
    task    - task to be signalled.
    signals - a set of signals to be submitted.

RESULTS
    the signal will show up in the signalled task's
    context.

EXAMPLE
    /* submit the (predefined) abortion signal: */
    TSignal(task, TTASK_SIG_ABORT);

NOTES
    it is valid to apply this function to both the caller's
    own task as well as to foreign tasks.

SEE ALSO
    TSetSignal, TWait
```

1.6 TSetSignal

```
NAME
    TSetSignal - set and get a task's signals.

SYNOPSIS
    oldsignals = TSetSignal(task, newsignals, sigmask);
                            TAPTR TUINT       TUINT

FUNCTION
    set (and get) task's signals state

INPUTS
    task       - task
    newsignals - new set of signals
    sigmask    - signal bits to be affected

EXAMPLES
    /* get the current state of all signals, but do not modify them */
    signals = TSetSignal(task, 0, 0);

    /* clear the pre-defined abortion signal */
    TSetSignal(task, 0, TTASK_SIG_ABORT);

NOTES
    it is valid to apply this function to the caller's own task as
    well as to another task. note, however, that the results may be
    confusing when a foreign context is being addressed: with this
    function it would be possible to wakeup a foreign task with the
    affecting signals being removed from its current signal state.

SEE ALSO
    TSignal, TWait
```

1.7 TWait

```
NAME
    TWait - wait for a set of signals.

SYNOPSIS
    signals = TWait(task, sigmask)
    TUINT           TAPTR TUINT

FUNCTION
    suspend task until one or more of the specified
    signals arrive. those bits will be cleared from
    the task's context when the function returns.

INPUTS
    task    - task, this MUST refer to the caller's context
    sigmask - mask of signals to wait for. if sigmask is 0,
              this function will return immediately.
RESULTS
    signals - signals that caused returning

NOTES
    if applied not to the caller’s own task, the results are
    entirely undefined, and it will likely break your software.

SEE ALSO
    TTimedWait, TWaitPort, TSignal
```

1.8 TTimedWait

```
    NAME
        TTimedWait - wait for a set of signals, with timeout

    SYNOPSIS
        signals = TTimedWait(task, sigmask, timeout)
        TUINT                TAPTR TUINT    TTIME*

    FUNCTION
        suspend task to wait for a set of signals, or for a timeout.
        any signals causing this function to return will be returned
        to the caller and cleared from the task’s set of signals. if
        a timeout caused the return, the return value will be 0.
        if timeout is TNULL or (timeout->sec and timeout->usec) are
        zero, this function is equivalent to TWait.

    INPUTS
        task    - task, this MUST refer to the caller’s context
        sigmask - mask of signals to wait for
        timeout - pointer to a TTIME specifier

    RESULTS
        signals - signals that caused returning, or 0 if timeout

    NOTES
        if applied to not the caller’s own task, the results are
        entirely undefined, and it will likely break your software.

    SEE ALSO
        TWait, TWaitPort, TSignal
```

1.9 TInitLock

```
    NAME
        TInitLock - initialize a task lock

    SYNOPSIS
        success = TInitLock(task, lock,  tags);
        TBOOL               TAPTR TLOCK* TTAGITEM*
```

```
FUNCTION
    initialize a task lock structure. a task lock is an
    atomic cross-task protection mechanism. after initialization,
    the object has no owner and is in unlocked state.

INPUTS
    task    - caller's own task.
    lock    - pointer to a TLOCK structure.
    tags    - pointer to an array of tag items.

TAGS
    none defined yet.

RESULTS
    success - TTRUE if initialization was successful, else TFALSE

NOTES
    a lock is destroyed with a call to TDestroy. results are
    undefined if a lock is destroyed in locked state. any call to
    TLock per calling context must be empaired with exactly
    one matching call to TUnlock.

SEE ALSO
    TLock, TUnlock, TDestroy
```

1.10 TLock

```
NAME
    TLock - gain exclusive access to a task lock.

SYNOPSIS
    TLock(lock);
          TLOCK*

FUNCTION
    gain exclusive access to a task lock. if another task is
    currenty holding the lock, the caller will block until the
    lock is released. if no other task holds the lock, this function
    will return immediately with exclusive access to the lock.

    this function is recurisve (or 'nesting'), i.e. it may be called
    again in the caller's context when the lock is already held in
    the caller's context. in that case an internal counter is
    increased, and this function will return immediately. each call
    per context must be empaired with exactly one matching call
    to TUnlock, which will decrease the counter. finally, when
    the counter reaches zero, the lock is actually released.

INPUTS
    lock    - pointer to a TLOCK structure, initialized with
              TInitLock

RESULTS
```

```
        none

    SEE ALSO
        TUnlock, TInitLock
```

1.11 TUnlock

```
    NAME
        TUnlock - release access to a task lock.

    SYNOPSIS
        TUnlock(lock);
                TLOCK*

    FUNCTION
        release access to a task lock, which was previously obtained with
        a call to TLock. see the function description there.

    INPUTS
        lock    - pointer to a TLOCK structure, initialized with
                  TInitLock

    RESULTS
        none

    SEE ALSO
        TLock, TInitLock
```

1.12 TCreatePort

```
    NAME
        TCreatePort - create a messageport.

    SYNOPSIS
        port = TCreatePort(task, tags)
        TPORT*             TAPTR TTAGITEM*

    FUNCTION
        allocate a signal from the given task, and create and initialize
        a port for message communication, which will belong to the
        task's context.

    INPUTS
        task    - task that will be the owner of the messageport.
                  this should be the caller's own task context.
        taglist - pointer to an array of tag items

    TAGS
        TTask_MMU, TAPTR
            pointer to a memory management unit to allocate the port
            structures from. default: the task's heap memory manager.

    RESULTS
        port    - messageport created, or TNULL on failure.

    NOTES
        - a port is destroyed with a call to TDestroy.

        - currently (v0.3) it is valid to create a messageport for a
          foreign task, but this should be rarely ever needed, and
          may result in a confusing application design. do not rely
          on this. future versions might limit this function strictly
          to the caller's own task context.

    SEE ALSO
        TWaitPort, TPutMsg, TPutReplyMsg, TDestroy
```

1.13 TWaitPort

```
    NAME
        TWaitPort - wait for a port to be non-empty

    SYNOPSIS
        TWaitPort(msgport)
                  TPORT*

    FUNCTION
        suspend a messageport's owner task until a message is
        present at its message queue. when a message is already
        present, return immediately.

    INPUTS
        msgport  - messageport. this port must be owned by the
                   caller's context.

    RESULTS
        none

    NOTES
        if the port does not belong to to the caller's own task context,
        the results are entirely undefined, and it will likely break
        your software.

    SEE ALSO
        TCreatePort, TWait, TGetMsg
```

1.14 TTimeDelay

```
    NAME
        TTimeDelay - sleep
```

```
SYNOPSIS
    TTimeDelay(task, time)
               TAPTR TTIME*

FUNCTION
    suspend the caller's task and sleep for the specified time.

INPUTS
    task    - task handle referring to the caller's context.
    time    - time structure

SEE ALSO
    TTimeQuery, TCreateTask
```

1.15 TTimeQuery

```
NAME
    TTimeQuery - query task timer

SYNOPSIS
    TTimeQuery(task, time)
               TAPTR TTIME*

FUNCTION
    this function queries a task's inbuilt timer and inserts the
    time elapsed since task creation into the specified time
    structure.

INPUTS
    task    - task handle to query.
    time    - time structure.

NOTES
    - a task's timer is initialized to zero when its task is created,
      therefore it measures the task's lifetime.

    - it is valid to query a foreign task's timer, i.e. the task
      handle does not need to refer to the caller's context.

SEE ALSO
    TTimeReset, TTimeDelay, TCreateTask
```

1.16 TTimeReset

```
NAME
    TTimeReset - reset task timer

SYNOPSIS
    TTimeReset(task)
```

```
                   TAPTR

    FUNCTION
        this function resets the given task's inbuilt timer to zero.

    INPUTS
        task    - task to reset

    SEE ALSO
        TTimeQuery, TTimeDelay, TCreateTask
```

1.17 TGetRandomSeed

```
    NAME
        TGetRandomSeed - get a seed value

    SYNOPSIS
        seed = TGetRandomSeed(task)
        TUINT                 TAPTR

    FUNCTION
        generate a random seed number.

    INPUTS
        task - task handle to query

    RESULTS
        seed - seed value for random number generation

    NOTES
        currently (v0.3) a seed number is generated from a task's
        individual timer, but the quality of this value may differ
        on different hosting environments, and may not be sufficient
        for advanced purposes, such as crypto key generation.

    SEE ALSO
        TGetRandom, TTimeQuery, TCreateTask
```

1.18 TTaskAlloc

```
    NAME
        TTaskAlloc - allocate memory from a task

    SYNOPSIS
        mem = TTaskAlloc(task, size)
        TAPTR            TAPTR TUINT

    FUNCTION
        allocate memory from a task's inbuilt heap memory manager.
```

```
INPUTS
    task - task handle to allocate from
    size - size of the requested block of memory [bytes]

RESULTS
    mem  - pointer to memory, or TNULL if memory exhausted.

NOTES
    - a task's heap memory manager implements thread-safety and
      cleanup handling by default. unless you do not specify a
      user MMU upon task creation, you may safely allocate from
      foreign tasks, and allocations will be freed automatically
      when their respective task exits.

    - this function is currently (v0.3) being implemented as
      a macro, redirecting the call to TMMUAlloc on a task's
      heap MMU.

SEE ALSO
    TTaskFree, TTaskAlloc0, TTaskRealloc, TTaskGetSize,
    TCreateTask
```

1.19 TTaskAlloc0

```
NAME
    TTaskAlloc0 - allocate blank memory from a task

SYNOPSIS
    mem = TTaskAlloc0(task, size)
    TAPTR             TAPTR TUINT

FUNCTION
    allocate blank memory from a task's inbuilt heap memory manager,
    i.e. the allocated block will be cleared with zero-bytes.

INPUTS
    task - task handle to allocate from
    size - size of the requested block of memory [bytes]

RESULTS
    mem  - pointer to memory, or TNULL if memory exhausted.

NOTES
    - see annotations for TTaskAlloc.

    - this function is currently (v0.3) being implemented as
      a macro, redirecting the call to TMMUAlloc0 on a task's
      heap MMU.

SEE ALSO
    TTaskFree, TTaskAlloc
```

1.20 TTaskFree

```
NAME
    TTaskFree - return memory to a task.

SYNOPSIS
    TTaskFree(task, mem)
              TAPTR TAPTR

FUNCTION
    return an allocation to a task's heap memory manager.

INPUTS
    task - task handle to allocate from
    mem  - pointer to an allocation made from a task

RESULTS
    none

NOTES
    - see annotations for TTaskAlloc.

    - this function is currently (v0.3) being implemented as
      a macro, redirecting the call to TMMUFree on a task's
      heap MMU.

SEE ALSO
    TTaskAlloc
```

1.21 TTaskRealloc

```
NAME
    TTaskRealloc - realloc an allocation from a task

SYNOPSIS
    newmem = TTaskRealloc(task, oldmem, newsize)
    TAPTR                 TAPTR TAPTR   TUINT

FUNCTION
    reallocate an allocation previously made from
    a task's heap memory manager.

    when the oldmem argument is TNULL, this function tries to
    allocate a new block of the given newsize. when newsize is
    zero and oldmem is given, the block will be freed, and TNULL
    will be returned. when oldmem is TNULL and newsize is zero,
    this function returns TNULL.

INPUTS
    task    - task handle
    oldmem  - pointer to an allocation from the task
    newsize - new size for the reallocated block of memory
```

```
RESULTS
    newmem  - pointer to memory being reallocated, or TNULL.

NOTES
    - reallocation may require that the given block of memory
      needs to be moved in memory, i.e. pointers to this area
      may become invalid.

    - see annotations for TTaskAlloc.

    - this function is currently (v0.3) being implemented as
      a macro, redirecting the call to TMMURealloc on a task's
      heap MMU.

SEE ALSO
    TTaskAlloc
```

1.22 TTaskGetSize

```
NAME
    TTaskGetSize - get size of an allocation from a task.

SYNOPSIS
    size = TTaskGetSize(task, mem)
    TUINT               TAPTR TAPTR

FUNCTION
    return the size of an allocation previously made from
    a task's heap memory manager.

INPUTS
    task - task handle
    mem  - pointer to an allocation made from the task

RESULTS
    size - size of the allocation [bytes]

NOTES
    - see annotations for TTaskAlloc.

    - this function is currently (v0.3) being implemented as
      a macro, redirecting the call to TMMUGetSize on a task's
      heap MMU.

SEE ALSO
    TTaskAlloc
```

1.23 TTaskAllocMsg

```
NAME
    TTaskAllocMsg - allocate a message.

SYNOPSIS
    msg = TTaskAllocMsg(task, size)
    TAPTR               TAPTR TUINT

FUNCTION
    allocate a message of the given size from a task.

INPUTS
    task  - task handle
    size  - size of the message [bytes]

RESULTS
    msg   - pointer to message buffer, or TNULL if out of memory

NOTES
    - the message size can be queried with TGetMsgAttrs.

    - this function is currently (v0.3) being implemented as
      a macro, redirecting the call to TMMUAlloc on a task's
      message MMU.

SEE ALSO
    TFreeMsg, TReplyMsg, TAckMsg, TDropMsg, TGetMsgAttrs,
    TMMUAlloc, TSendMsg
```

1.24 TTaskBaseTask

```
NAME
    TTaskBaseTask - get base task handle.

SYNOPSIS
    basetask = TTaskBaseTask(task)
    TAPTR                    TAPTR

FUNCTION
    return a pointer to the root task context of a TEKlib framework.
    the pointer to the base task handle is carried in each of its
    childs. it is also valid to apply this function to the basetask
    itself.

INPUTS
    task      - a task handle

RESULTS
    basetask  - pointer to the application framework's base task

NOTES
    this function is currently (v0.3) being implemented as a macro.

SEE ALSO
```

```
    TCreateTask
```

1.25 TTaskHeapMMU

```
NAME
    TTaskHeapMMU - get a task's heap memory manager.

SYNOPSIS
    heapmmu = TTaskHeapMMU(task)
    TAPTR                  TAPTR

FUNCTION
    return a pointer to a task's heap memory manager.

INPUTS
    task      - task handle

RESULTS
    heapmmu   - pointer to the task's heap MMU.

NOTES
    this function is currently (v0.3) being implemented as a macro.

SEE ALSO
    TCreateTask, TTaskAlloc, TTaskMsgMMU, TInitMMU
```

1.26 TTaskMsgMMU

```
NAME
    TTaskMsgMMU - get a task's message memory manager.

SYNOPSIS
    msgmmu = TTaskMsgMMU(task)
    TAPTR                TAPTR

FUNCTION
    return a pointer to a task's message memory manager.

INPUTS
    task    - task handle

RESULTS
    msgmmu  - pointer to the task's message MMU.

NOTES
    this function is currently (v0.3) being implemented as a macro.

SEE ALSO
    TCreateTask, TTaskAllocMsg, TTaskHeapMMU, TInitMMU
```

1.27 TTaskGetData

```
NAME
    TTaskGetData - get a task's userdata pointer.

SYNOPSIS
    userdata = TTaskGetData(task)
    TAPTR                   TAPTR

FUNCTION
    return a pointer to a task's userdata.

INPUTS
    task     - task handle

RESULTS
    userdata - pointer to the task's userdata.

NOTES
    this function is currently (v0.3) being implemented as a macro.

SEE ALSO
    TTaskSetData, TCreateTask
```

1.28 TTaskSetData

```
NAME
    TTaskSetData - change a task's userdata pointer.

SYNOPSIS
    TTaskSetData(task, userdata)
                 TAPTR TAPTR

FUNCTION
    change a task's userdata pointer.

INPUTS
    task     - task handle
    userdata - arbitrary pointer to user data.

NOTES
    - if you want to modify and query a task's userdata pointer from
      different task contexts during a task's lifetime, you will
      probably need to implement a locking mechanism to ensure data
      integrity. you might find it more reliable to leave the primary
      userdata pointer unmodified, and reference userdata indirectly:

      struct taskuserdata
      {
          TLOCK lock;
          TAPTR userdata;
      };
```

```
        TVOID taskfunc(TAPTR task)
        {
            struct taskuserdata *d = TTaskGetData(task);
            TLock(&d->lock);
            /* set and get and operate on d->userdata pointer safely */
            TUnlock(&d->lock);
        }

        you can, however, safely set and get a task's userdata pointer
        inside a task's init function, because the newly created context
        is unknown to other task contexts at this time. there is no
        locking required in a task's init function.

      - this function is currently (v0.3) being implemented as a macro.

  SEE ALSO
      TTaskGetData, TCreateTask, TInitLock
```

1.29 TTaskPort

```
  NAME
      TTaskPort - get a task's messageport.

  SYNOPSIS
      port = TTaskPort(task)
      TPORT*           TAPTR

  FUNCTION
      return a pointer to a task's messageport.

  INPUTS
      task     - task handle

  RESULTS
      port     - pointer to task's messageport

  NOTES
      this function is currently (v0.3) being implemented as a macro.

  SEE ALSO
      TCreateTask, TCreatePort
```

1.30 TFreeMsg

```
  NAME
      TFreeMsg - free a message.

  SYNOPSIS
      TFreeMsg(msg)
               TAPTR
```

```
FUNCTION
    free a message and return its memory to the message memory
    manager it has been allocated from.

    this function may be applied only when a message was allocated
    but never sent, or when it has been sent as a two-way message
    with TPutReplyMsg, and returned to a replyport.

    one-way messages sent with TPutMsg are freed transparently
    with either TAckMsg or TReplyMsg at the destination
    endpoint.

INPUTS
    msg      - message to be freed.

NOTES
    this function is currently (v0.3) being implemented as a macro.

SEE ALSO
    TTaskAllocMsg, TDropMsg, TReplyMsg, TAckMsg, TPutMsg,
    TPutReplyMsg, TSendMsg
```

1.31 TPutMsg

```
NAME
    TPutMsg - send a one-way message.

SYNOPSIS
    TPutMsg(msgport, msg)
            TPORT*   TAPTR

FUNCTION
    put a one-way message to a messageport. one-way messages
    do not return to the sender. this function never blocks.

    messages sent to messageports in the caller's local address
    space are reliable, whereas messages put to remote ports are
    not. you may only assume that a one-way message has been
    successfully delivered to a remote port when you receive a
    corresponding reply, which in some way needs to be defined
    elsewhere in your individual protocols.

    when you don't know whether the addressed messageport
    is in your local address space or not, you must consider
    message delivery with this function to be unreliable.

INPUTS
    msgport    - messageport to be addressed.
    msg        - message to be sent.

NOTES
    messages can be sent reliably with TPutReplyMsg.
```

```
SEE ALSO
    TPutReplyMsg, TGetMsg, TAckMsg, TReplyMsg, TDropMsg,
    TTaskAllocMsg, TFreeMsg, TSendMsg, TCreatePort
```

1.32 TPutReplyMsg

```
NAME
    TPutReplyMsg - send a two-way message.

SYNOPSIS
    TPutReplyMsg(msgport, replyport, msg)
                 TPORT*   TPORT*     TAPTR

FUNCTION
    put a two-way message to a messageport, with a reply
    or acknowledgement being delivered to the given replyport.
    two-way messages always return to the sender. this function
    never blocks.

    message delivery to a messageport in local address space
    is defined to be reliable, and will always succeed. message
    delivery over unreliable transmission paths (such as TCP/IP
    network connections), on the other hand, may always fail.

    this function ensures that the sender will be informed about
    a message's fate, regardless whether the addressed port is
    in local address space or not.

    messages that could not be delivered (or failed to return)
    over an unreliable connection will appear on the given
    replyport with their status set to TMSG_STATUS_FAILED.
    successful delivery will be indicated with a status set to
    TMSG_STATUS_REPLIED or TMSG_STATUS_ACKD (depending on the
    reply method). the message status can be queried with
    TGetMsgAttrs.

    after the message arrived at its replyport, it usually needs
    be freed with TFreeMsg. it is possible, however, to reuse
    a message.

INPUTS
    msgport    - messageport to be addressed.
    replyport  - replyport to which the message will be returned.
    msg        - message to be sent.

SEE ALSO
    TPutMsg, TGetMsg, TAckMsg, TReplyMsg, TDropMsg,
    TTaskAllocMsg, TFreeMsg, TSendMsg, TCreatePort
```

1.33 TGetMsg

```
NAME
    TGetMsg - get message.

SYNOPSIS
    msg = TGetMsg(msgport)
    TAPTR          TPORT*

FUNCTION
    unlink the next pending message from a messageport's queue
    and return it to the caller. this function does not block.

    a message's status and other attributes can be queried with
    TGetMsgAttrs.

INPUTS
    msgport    - messageport to get next message from.

RESULTS
    msg        - next pending message, or TNULL if the
                 messageport queue was empty.

SEE ALSO
    TPutMsg, TPutReplyMsg, TAckMsg, TReplyMsg,
    TTaskAllocMsg, TFreeMsg, TCreatePort
```

1.34 TAckMsg

```
NAME
    TAckMsg - acknowledge message.

SYNOPSIS
    TAckMsg(msg)
            TAPTR

FUNCTION
    acknowledge a two-way message to its sender, i.e. return
    it to its sender's replyport.

    it is safe, however, to apply this function to one-way
    messages as well; if the message was sent without a reply
    or acknowledgement expected, it will be silently freed by
    this function.

    when a message is returned with this function, the sender
    must not rely on any modifications made inside the message body.
    if you want to modify data inside the message and send its
    modified contents back to the sender, you should use TReplyMsg
    instead.

INPUTS
    msg     - message to be acknowledged to its sender
              (or to be freed, transparently)
```

```
SEE ALSO
    TReplyMsg, TFreeMsg, TDropMsg, TPutMsg, TPutReplyMsg,
    TFreeMsg, TTaskAllocMsg, TSendMsg, TCreatePort
```

1.35 TReplyMsg

```
NAME
    TReplyMsg - reply message.

SYNOPSIS
    TReplyMsg(msg)
              TAPTR

FUNCTION
    reply a two-way message to its sender, i.e. return its entire
    contents back to its sender's replyport.

    it is safe, however, to apply this function to one-way
    messages as well; if the message was sent without a reply
    or acknowledgement expected, it will be silently freed by
    this function.

    use this function for transferring a modified message body
    back to its sender. if the message was not modified and it is
    only required to inform the sender that it has been delivered,
    then you should prefer TAckMsg.

INPUTS
    msg     - message to be replied to its sender.
              (or to be freed, transparently)

SEE ALSO
    TAckMsg, TFreeMsg, TDropMsg, TPutMsg, TPutReplyMsg,
    TFreeMsg, TTaskAllocMsg, TSendMsg, TCreatePort
```

1.36 TDropMsg

```
NAME
    TDropMsg - abandon a message.

SYNOPSIS
    TDropMsg(msg)
             TAPTR

FUNCTION
    abandon a two-way message, i.e. return it to its replyport with
    the message status set to TMSG_STATUS_FAILED. this function is
    not guaranteed to return any modifications made inside the
    message body, it will only indicate failure.
```

```
        it is safe to apply this function to one-way messages as well;
        if the message was sent without a reply or acknowledgement
        expected, it will be silently freed by this function.

    INPUTS
        msg     - message to be abandoned.

    NOTES
        currently (v0.3), if applied to a remote messageport, this
        function will not only abandon a single message, but the entire
        underlying socket proxy. all messages sent after the one being
        dropped will fail on this network connection, and pending replies
        will fail after their respective timeout.

    SEE ALSO
        TAckMsg, TReplyMsg, TFreeMsg, TPutMsg, TPutReplyMsg,
        TTaskAllocMsg, TSendMsg, TCreatePort
```

1.37 TSendMsg

```
    NAME
        TSendMsg - send a message, synchronized

    SYNOPSIS
        replymsg = TSendMsg(task, msgport, msg)
        TAPTR               TAPTR TPORT*   TAPTR

    FUNCTION
        this function sends a message two-way, synchronized, and waits
        for either a reply to return, or for the messageport's timeout.
        this is currently the only messaging function that may block.

        the return value will be either set to msg, indicating that the
        message has been sent and acknowledged/replied successfully,
        or TNULL, when the message could not be sent or did not return
        within a remote msgport's timeout.

    INPUTS
        task     - task, must refer to the caller's context.
        msgport  - msgport to address.
        msg      - message to be sent.

    RETURNS
        replymsg - will be set to msg when the message was sent and
                   returned successfully, otherwise TNULL.

    SEE ALSO
        TAckMsg, TReplyMsg, TDropMsg, TFreeMsg, TPutMsg,
        TPutReplyMsg, TTaskAllocMsg, TCreatePort, TFindSockPort
```

1.38 TGetMsgAttrs

```
NAME
    TGetMsgAttrs - query message attributes.

SYNOPSIS
    numattr = TGetMsgAttrs(msg,  tags)
    TUINT                  TAPTR TTAGITEM*

FUNCTION
    this function queries a given set of attributes from a message.
    the attributes will be filled into the taglist's respective
    variable pointers, and the number of attributes successfully
    retrieved will be returned to the caller.

INPUTS
    msg     - message to be queried.
    tags    - pointer to an array of tagitems.

RESULTS
    numattr - number of attributes filled into their respective
              variable pointers.

TAGS
    TMsg_Size, TUINT *
        the variable being pointed to by the tag value will be filled
        with the size of the message in bytes.

    TMsg_Status, TUINT *
        the variable being pointed to by the tag value will be filled
        with the message status, which can be

            TMSG_STATUS_UNDEFINED - message was never sent.

            TMSG_STATUS_SENT      - the message has been sent
                                    successfully.

            TMSG_STATUS_FAILED    - the message could not be sent or
                                    failed to return within a given
                                    timeout period.

            TMSG_STATUS_REPLIED   - the message has been replied
                                    successfully, and returned to the
                                    sender.

            TMSG_STATUS_ACKD      - the message has been acknowledged
                                    successfully, and returned to the
                                    sender.

    TMsg_Sender, TSTRPTR *
        the variable being pointed to by the tag value will be set to
        a pointer to a string, which will contain a sender
        messageport's unique name. this name is currently (v0.3) being
        composed from the sender's host and port number, such as
        "192.168.0.77:32452". for messages originating from local
        address space, this pointer will be set to TNULL.
```

```
            warning: this string pointer is no longer valid after the
            message has been handed over to TReplyMsg, TAckMsg,
            TFreeMsg, or TDropMsg.

        TMsg_SenderHost, TSTRPTR *
            the variable being pointed to by the tag value will be set to
            a pointer to a string containing the sender's host, e.g.
            "192.168.0.77". for messages originating from local address
            space, this pointer will be set to TNULL.

            warning: this string pointer is no longer valid after the
            message has been handed over to TReplyMsg, TAckMsg,
            TFreeMsg, or TDropMsg.

        TMsg_SenderPort, TUINT *
            the variable being pointed to by the tag value will be set to
            the sender messageport's internet port number, which may range
            from 0 to 65535. for messages originating from local address
            space, the portnumber will be set to 0xffffffff.

    NOTES
        it would be unwise to assume a specific format for the strings
        returned by TMsg_Sender or TMsg_SenderHost. currently (v0.3), the
        string format returned will reflect the ipv4 addressing scheme.

    SEE ALSO
        TGetMsg, TPutMsg, TPutReplyMsg, TAckMsg, TReplyMsg,
        TDropMsg, TFreeMsg, TTaskAllocMsg, TSendMsg, TCreatePort
```

1.39 TGetMsgStatus

```
    NAME
        TGetMsgStatus - get message status.

    SYNOPSIS
        status = TGetMsgStatus(msg)
        TUINT                  TAPTR

    FUNCTION
        get a message's delivery status.

    INPUTS
        msg     - message to be queried.

    RESULTS
        status  - message's delivery status, which can be

                  TMSG_STATUS_UNDEFINED - message was never sent.

                  TMSG_STATUS_SENT      - the message has been sent
                                          successfully.

                  TMSG_STATUS_FAILED    - the message could not be sent or
```

```
                                         failed to return within a given
                                         timeout period.

                  TMSG_STATUS_REPLIED   - the message has been replied
                                         successfully, and returned to the
                                         sender.

                  TMSG_STATUS_ACKD      - the message has been acknowledged
                                         successfully, and returned to the
                                         sender.

    NOTES
        this function is currently being (v0.3) implemented as a macro.

    SEE ALSO
        TGetMsgAttrs, TGetMsgSize
```

1.40 TGetMsgSize

```
    NAME
        TGetMsgSize - get message size.

    SYNOPSIS
        size = TGetMsgSize(msg)
        TUINT              TAPTR

    FUNCTION
        get message size.

    INPUTS
        msg     - message to be queried.

    RESULTS
        size    - size of the message in bytes.

    NOTES
        this function is currently being (v0.3) implemented as a macro.

    SEE ALSO
        TGetMsgAttrs, TGetMsgStatus
```

1.41 TAddHead

```
    NAME
        TAddHead - add a node at the head of a list.

    SYNOPSIS
        TAddHead(list,  node)
                 TLIST* TNODE*
```

```
FUNCTION
    add a node at the head of a doubly linked list.

INPUTS
    list - pointer to a list header.
    node - pointer to a node to be inserted.

SEE ALSO
    TAddTail, TInitList
```

1.42 TAddTail

```
NAME
    TAddTail - add a node at the tail of a list.

SYNOPSIS
    TAddTail(list,  node)
             TLIST* TNODE*

FUNCTION
    add a node at the tail of a doubly linked list.

INPUTS
    list - pointer to a list header.
    node - pointer to a node to be inserted.

SEE ALSO
    TAddHead, TInitList
```

1.43 TInsert

```
NAME
    TInsert - insert a node into a list.

SYNOPSIS
    TInsert(list,  node,  listnode)
            TLIST* TNODE* TNODE *

FUNCTION
    insert a node into a doubly linked list after the given
    listnode. if listnode == TNULL, this function is equivalent
    to TAddFirst().

INPUTS
    list - pointer to a list header.
    node - pointer to a node to be inserted.
    listnode - pointer to a node after which to insert.

SEE ALSO
    TInitList, TRemove, TAddHead
```

1.44 TRemove

```
NAME
    TRemove - unlink a node from a list.

SYNOPSIS
    TRemove(node)
            TNODE*

FUNCTION
    remove, i.e. unlink a node from whatever list it might
    be linked into.

INPUTS
    list - pointer to a list header.
    node - pointer to a node to be removed.

NOTES
    calling this function with a node not being part of a list
    may be fatal.

SEE ALSO
    TRemHead, TRemTail, TInitList
```

1.45 TRemHead

```
NAME
    TRemHead - unlink the first node of a list.

SYNOPSIS
    node = TRemHead(list)
    TNODE*          TLIST*

FUNCTION
    remove, i.e. unlink and return the first node from
    a doubly linked list.

INPUTS
    list - pointer to a list header.

RESULTS
    node - pointer to the node that has been removed, or
           TNULL when the list was empty.

SEE ALSO
    TRemTail, TRemove, TInitList
```

1.46 TRemTail

```
NAME
    TRemTail - unlink the last node of a list.

SYNOPSIS
    node = TRemTail(list)
    TNODE*          TLIST*

FUNCTION
    remove, i.e. unlink and return the last node from
    a doubly linked list.

INPUTS
    list - pointer to a list header.

RESULTS
    node - pointer to the node that has been removed, or
           TNULL when the list was empty.

SEE ALSO
    TRemHead, TRemove, TInitList
```

1.47 TSeekNode

```
NAME
    TSeekNode - seek node.

SYNOPSIS
    node = TSeekNode(node,  numsteps)
    TNODE*           TNODE* TINT

FUNCTION
    starting from node, seek by the given number of steps
    either forwards (steps > 0) or backwards (steps < 0).
    when steps == 0, the current node is returned. when the
    list is seeked past end or before start, TNULL will be
    returned.

INPUTS
    node  - pointer to a node inside a list.
    steps - number of steps to be seeked.

RESULTS
    node - pointer to the node reached, or TNULL.

SEE ALSO
    TInitList
```

1.48 TInitList

```
NAME
    TInitList - prepare a list header structure.

SYNOPSIS
    TInitList(list)
              TLIST*

FUNCTION
    prepare a list header structure. the list
    will be empty and ready for usage.

INPUTS
    list - pointer to an uninitialized list structure.

NOTE
    this function is currently (v0.3) being implemented as
    a macro.

SEE ALSO
    TAddHead, TAddTail, TInsert, TRemove, TRemHead,
    TRemTail, TSeekNode, TFirstNode, TLastNode,
    TListEmpty
```

1.49 TFirstNode

```
NAME
    TFirstNode - get first node of a list.

SYNOPSIS
    node = TFirstNode(list)
    TNODE*            TLIST*

FUNCTION
    return the first node in a list, or TNULL when the list
    is empty.

INPUTS
    list - pointer to a list header.

RESULTS
    node - pointer to the first node in a list, or TNULL.

NOTE
    this function is currently (v0.3) being implemented as
    a macro.

SEE ALSO
    TLastNode, TListEmpty, TInitList
```

1.50 TLastNode

```
NAME
    TLastNode - get last node of a list.

SYNOPSIS
    node = TLastNode(list)
    TNODE*           TLIST*

FUNCTION
    return the last node in a list, or TNULL when the list
    is empty.

INPUTS
    list - pointer to a list header.

RESULTS
    node - pointer to the last node in a list, or TNULL.

NOTE
    this function is currently (v0.3) being implemented as
    a macro.

SEE ALSO
    TFirstNode, TListEmpty, TInitList
```

1.51 TListEmpty

```
NAME
    TListEmpty - test if a list is empty.

SYNOPSIS
    isempty = TListEmpty(list)
    TBOOL                TLIST*

FUNCTION
    test if a list is empty.

INPUTS
    list    - pointer to a list header.

RESULTS
    isempty - boolean, TTRUE when there are no nodes linked to
              the list.

NOTE
    this function is currently (v0.3) being implemented as
    a macro.

SEE ALSO
    TFirstNode, TInitList
```

1.52 TGetTagValue

```
NAME
    TGetTagValue - get tag value from a tag list

SYNOPSIS
    value = TGetTagValue(tag, defaultvalue, taglist)
    TTAG                 TTAG TTAG          TTAGITEM*

FUNCTION
    parse a list of tag items and return the matching
    tag value. if the specified tag is not contained in
    the list, return the default value.

INPUTS
    tag          - tag to be queried.
    defaultvalue - default tag value.
    taglist      - pointer to a list of tag items.

RESULTS
    value - the value associated with the queried tag, if found
            in the taglist, otherwise the default value.

SEE ALSO
    TGetTagArray, TInitTags, TAddTag
```

1.53 TGetTagArray

```
NAME
    TGetTagArray - get an array of tag values from a tag list

SYNOPSIS
    numtags = TGetTagArray(taglist,  tagarray)
    TUINT                  TTAGITEM* TTAG*

FUNCTION
    this function parses an array of tag items and a taglist, and
    transfers the values of all matching tags from the taglist into
    the variables referenced by pointers in the tag array. both
    the tag array and the taglist must be concluded with TTAG_DONE.
    the number of tags that have been retrieved will be returned to
    the caller.

EXAMPLE
    TTAG one = 1; two = 2; three = 3;  /* default values */

    num = TGetTagArray(taglist,
            MYTAG_One, (TTAG) &one,
            MYTAG_Two, (TTAG) &two,
            MYTAG_Three, (TTAG) &three,
            TTAG_DONE);

INPUTS
```

```
        taglist  - pointer to a list of tag items.
        tagarray - pointer to an array of pairs of
                   tag and variable pointer each.

    RESULTS
        numtags - number of tags that have been retrieved
                  from the taglist, and inserted to their
                  respective variables.

    SEE ALSO
        TGetTagValue, TInitTags, TAddTag
```

1.54 TInitTags

```
    NAME
        TInitTags - init an array of tagitems.

    SYNOPSIS
        TInitTags(taglist)
                  TTAGITEM*

    FUNCTION
        prepare an array of tagitems to be filled with tag
        attributes, using TAddTag.

    INPUTS
        taglist  - pointer to an array of tag items

    NOTE
        this function is currently (v0.3) being implemented as
        a macro.

    SEE ALSO
        TAddTag, TGetTagValue, TGetTagArray
```

1.55 TAddTag

```
    NAME
        TAddTag - add a tag/value pair to a taglist.

    SYNOPSIS
        TAddTag(taglist,  tag, value)
                TTAGITEM* TTAG TTAG

    FUNCTION
        add a single tag/value pair to a list of tag items.

        your taglist must be dimensioned to contain at least
        one more item than the number of items being added
        with this function.
```

```
INPUTS
    taglist  - pointer to an array of tag items
    tag      - tag identifier
    value    - tag value

NOTE
    - this function is currently (v0.3) being implemented as
      a macro.

    - this is a convenience macro. it may save a few keystrokes,
      but it is suboptimal. it is quicker to fill a tag list
      manually.

SEE ALSO
    TInitTags, TGetTagValue, TGetTagArray
```

1.56 TGetRandom

```
NAME
    TGetRandom - generate signed random number

SYNOPSIS
    random = TGetRandom(seed)
    TINT                TINT

FUNCTION
    generate a 32 bit pseudo random number, which will be
    computed from the seed value. the number returned will
    be in the range from -2147483648 to 2147483647.

    typically the returned number will be fed back to
    TGetRandom as the new seed value for the next number
    generation cycle.

EXAMPLE
    /* generate a random number from 0 to 343 */

    TINT seed, rand_value;
    rand_value = (seed = TGetRandom(seed)) % 344;

INPUTS
    seed   - a seed value for the number generator.

RESULTS
    random - a pseudo random number.

NOTES
    - the numbers generated by this function are not random.
      a number series is always fully determined by its initial
      seed value. the series only appears to be random in an
      arbitrary short range.

    - for useful random numbers the seed variable should be
```

```
        initialized with a hardly deterministic number.

SEE ALSO
    TGetRandomSeed
```

1.57 TMemCopy

```
NAME
    TMemCopy - copy a block of memory.

SYNOPSIS
    TMemCopy(source, dest, numbytes)
             TAPTR   TAPTR TUINT

FUNCTION
    copy a block of memory, i.e. the given number of bytes
    from source to dest.

INPUTS
    source   - source address
    dest     - destination address
    numbytes - number of bytes to copy

NOTES
    you may not rely on overlapping copies to work with
    this function.

SEE ALSO
    TMemCopy32, TMemFill
```

1.58 TMemFill

```
NAME
    TMemFill - fill a block of memory.

SYNOPSIS
    TMemFill(start, numbytes, fillval)
             TAPTR  TUINT     TUINT

FUNCTION
    fill a range of memory with a character fill value.

INPUTS
    start    - start address
    numbytes - number of bytes to fill
    fillval  - character to fill in

SEE ALSO
    TMemFill32, TMemCopy
```

1.59 TMemCopy32

```
NAME
    TMemCopy32 - copy a block of memory, aligned

SYNOPSIS
    TMemCopy32(source, dest, numbytes)
               TAPTR   TAPTR TUINT

FUNCTION
    copy a block of memory, i.e. the given number of bytes
    from source to dest. the source and destination address
    must be aligned to 32 bit boundaries in memory, and the
    number of bytes must be 32 bit aligned as well.

INPUTS
    source   - source address, 32bit aligned
    dest     - destination address, 32bit aligned
    numbytes - number of bytes to copy, 32bit aligned

NOTES
    you may not rely on overlapping copies to work with
    this function.

SEE ALSO
    TMemCopy, TMemFill32
```

1.60 TMemFill32

```
NAME
    TMemFill - fill a block of memory, aligned

SYNOPSIS
    TMemFill32(start, numbytes, fillval)
               TAPTR  TUINT     TUINT

FUNCTION
    fill a range of memory with a 32bit fill value. the
    start address and the number of bytes must be aligned
    to 32 bit.

INPUTS
    start    - start address, 32bit aligned
    numbytes - number of bytes to fill, 32bit aligned
    fillval  - 32bit value to fill in

SEE ALSO
    TMemFill, TMemCopy32
```

1.61 TInitMemHead

```
NAME
    TInitMemHead - initialize a memheader.

SYNOPSIS
    success = TInitMemHead(memhead,  mem,  size, tags)
    TBOOL                  TMEMHEAD* TAPTR TUINT TTAGITEM*

FUNCTION
    initialize a memheader. a memheader is a memory range
    descriptor that can be used for lowlevel allocation
    from a static block of memory.

INPUTS
    memhead - pointer to an uninitialized memheader structure
    mem     - pointer to a block of memory to be used as a
              static memory allocation pool
    size    - size of the memory block [bytes]
    tags    - pointer to an array of tag items

TAGS
    none defined yet

RESULTS
    success  - boolean indicating whether initialization
               was successful. TTRUE if the header is ready.

EXAMPLE
    /* setup a memheader at the beginning of a memory block */

    TUINT8 memory[100000];

    TInitMemHead((TMEMHEAD *) memory, memory + sizeof(TMEMHEAD),
        sizeof(memory) - sizeof(TMEMHEAD), TNULL);

    /* now ((TMEMHEAD *) memory) may be passed to functions like
       TStaticAlloc and TStaticRealloc. */

SEE ALSO
    TStaticAlloc, TStaticFree, TStaticRealloc,
    TStaticGetSize
```

1.62 TStaticAlloc

```
NAME
    TStaticAlloc - allocate memory from a static block of memory.

SYNOPSIS
    mem = TStaticAlloc(memhead,  size)
    TAPTR              TMEMHEAD* TUINT

FUNCTION
```

```
        allocate from a block of static memory, which is described by
        a memhead structure. returns a block of memory of the given size,
        or TNULL when the request could not be satisfied.

    INPUTS
        memhead - pointer to an initialized memheader structure
        size    - size of the request [bytes]

    RESULTS
        mem     - memory allocated, or TNULL, when there was no
                  block of memory of the requested size available.

    NOTES
        it is not allowed to pass TNULL or zero for either memhead
        or size. this function is designed for low overhead.

    SEE ALSO
        TInitMemHeadA(), TStaticFree, TStaticRealloc,
        TStaticGetSize
```

1.63 TStaticRealloc

```
    NAME
        TStaticRealloc - reallocate an allocation from static memory.

    SYNOPSIS
        newmem = TStaticRealloc(memhead,  oldmem, newsize)
        TAPTR                   TMEMHEAD* TAPTR   TUINT

    FUNCTION
        resize a block of memory from a static memory allocation to the
        specified size, and return a pointer to the resized block of
        memory, or TNULL when the memory block could not be resized.

        when a memory block is supplied, and newsize is zero, then
        the memory block will be returned to the static block of
        memory, and the result of this function is TNULL.

        when newsize is nonzero, and the memory block is TNULL, this
        function will try to allocate a new block of the given size.

        if mem is TNULL and size is zero, this function will return
        TNULL.

    INPUTS
        memhead - pointer to an initialized memheader structure
        oldmem  - pointer to a block of memory to be resized
        newsize - new size of the block [bytes]

    RESULTS
        mem     - resized (or freshly allocated) block of memory,
                  or TNULL.

    NOTES
```

```
        - it is not allowed to pass TNULL for the memhead argument.

        - reallocation may require that the given block of memory
          needs to be moved in memory, i.e. pointers to this area
          may become invalid.

    SEE ALSO
        TInitMemHeadA(), TStaticAlloc, TStaticFree,
        TStaticGetSize
```

1.64 TStaticFree

```
    NAME
        TStaticFree - return memory to a static block of memory.

    SYNOPSIS
        TStaticFree(memhead,  mem)
                    TMEMHEAD* TAPTR

    FUNCTION
        free a block of memory and return it to the static block
        of memory it was allocated from.

    INPUTS
        memhead - pointer to an initialized memheader structure
        mem     - pointer to a block of memory to be freed

    NOTES
        it is not allowed to pass TNULL or zero for either memhead
        or mem. this function is designed for low overhead.

    SEE ALSO
        TInitMemHeadA(), TStaticAlloc, TStaticRealloc,
        TStaticGetSize
```

1.65 TStaticGetSize

```
    NAME
        TStaticGetSize - get size of an allocation from static memory.

    SYNOPSIS
        size = TStaticGetSize(memhead,  mem)
        TUINT                 TMEMHEAD* TAPTR

    FUNCTION
        this function returns the size of an allocation made
        with TStaticAlloc or TStaticRealloc.

    INPUTS
        memhead - pointer to an initialized memheader structure
        mem     - previously allocated block of memory, or TNULL

    RESULTS
        size    - size of the allocation [bytes].

    NOTE
        it is not allowed to pass TNULL or zero for either memhead
        or mem. this function is designed for low overhead.

    SEE ALSO
        TInitMemHeadA(), TStaticAlloc, TStaticRealloc
```

1.66 TCreatePool

```
NAME
    TCreatePool - create pooled allocator.

SYNOPSIS
    pool = TCreatePool(mmu,  chunksize, thressize, tags)
    TAPTR              TAPTR TUINT      TUINT      TTAGITEM*

FUNCTION
    create and initialize a pooled memory allocator.

    pools can automatically expand and shrink on demand. many
    individual allocations may fit into chunks which are being
    maintained internally by the pooled allocator.

    there is no need to free individual allocations requested
    from a pooled allocator; they will be freed automatically
    when the pool is destroyed with TDestroy.

    chunksize is the size of new chunks to be allocated from
    a parent memory manager, when a new allocation cannot be
    satisfied from the current set of chunks.

    thressize is the maximum size of allocations that will be
    allocated from regular chunks. allocations larger than
    thressize will request new chunks of their own.

    pools created with 'dynamic growth' will automatically adapt
    their chunksize, and always allocate new chunks larger than
    required by a single allocation. TPoolRealloc will utilize
    this prefetch memory to allow rapidly growing reallocations
    with very few overhead. with dynamic growth enabled, chunksize
    divided by thressize will be used as the pool's prefetch ratio.

INPUTS
    mmu       - parent memory manager
    chunksize - size of chunks to be allocated from parent
                memory manager
    thressize - maximum size of allocations that will be
                requested from chunks of their own
    taglist   - pointer to an array of tag items
TAGS
    TMem_DynGrow, TBOOL
        when this argument is set to TTRUE, chunksize/threshold
        are interpreted as an initial ratio for dynamic pool
        growth. default: TTRUE

RESULTS
    pool - an initialized memory pool, or TNULL if something
           went wrong.

SEE ALSO
    TPoolAlloc, TPoolRealloc, TPoolFree, TPoolGetSize,
    TDestroy
```

1.67 TPoolAlloc

```
    NAME
        TPoolAlloc - allocate memory from a pool.

    SYNOPSIS
        mem = TPoolAlloc(pool, size)
        TAPTR            TAPTR TUINT

    FUNCTION
        allocate a block of memory of the given size from a pool.

   INPUTS
        pool - an object created with TCreatePool
        size - size of the allocation [bytes]

    RESULTS
        mem - pointer to a block of memory, or TNULL when the request
              could not be satisfied.

    SEE ALSO
        TCreatePool, TPoolFree, TPoolRealloc, TPoolGetSize
```

1.68 TPoolFree

```
    NAME
        TPoolFree - return memory to a pool.

    SYNOPSIS
        TPoolFree(pool, mem)
        void      TAPTR TAPTR

    FUNCTION
        return a block of memory to a pool.
```

```
INPUTS
    pool - a pooled allocator created with TCreatePool.
    mem  - pointer to a block of memory allocated with
           TPoolAllloc().

SEE ALSO
    TCreatePool, TPoolAlloc, TPoolRealloc, TPoolGetSize
```

1.69 TPoolRealloc

```
NAME
    TPoolRealloc - resize a block of memory in a pool.

SYNOPSIS
    mem = TPoolRealloc(pool, oldmem, size)
    TAPTR              TAPTR TAPTR   TUINT

FUNCTION
    resizes a memory block that was allocated from a pool to the
    specified size, and returns a valid pointer to the resized block
    of memory, or TNULL when the memory block could not be resized.

    when a memory block is passed, but the specified size is zero,
    the memory block will be returned to the pool, and the result of
    this function is TNULL.

    when a size is specified, and the memory block is TNULL, this
    function will try to allocate a new block of the given size.

    if mem is TNULL and size is zero, this function will return
    TNULL.

INPUTS
    pool    - an object created with TCreatePool.
    oldmem  - pointer to a block of memory to be resized.
    size    - new size of the memory block.

RESULTS
    mem     - resized (or freshly allocated) block of memory,
              or TNULL.

NOTES
    reallocation may require that the given block of memory
    needs to be moved in memory, i.e. pointers to this area
    may become invalid.

SEE ALSO
    TPoolAlloc, TPoolFree, TCreatePool, TPoolGetSize
```

1.70 TPoolGetSize

```
NAME
    TPoolGetSize - get size of an allocation from a pool.

SYNOPSIS
    size = TPoolGetSize(TAPTR pool, mem)
    TUINT               TAPTR       TAPTR

FUNCTION
    this function returns the size of an allocation made
    with TPoolAlloc or TPoolRealloc. if mem is TNULL,
    this function returns 0.

INPUTS
    mem  - previously allocated block of memory, or TNULL.

RESULTS
    size - size of the allocation [bytes].

SEE ALSO
    TPoolAlloc, TPoolRealloc
```

1.71 TInitMMU

```
NAME
    TInitMMU - initialize a memory management unit.

SYNOPSIS
    success = TInitMMU(mmu,  allocator, mmutype, tags)
    TBOOL              TMMU* TAPTR      TUINT    TTAGITEM*

FUNCTION
    initialize a TMMU structure and prepare it for being
    used as a memory management unit.

INPUTS
    mmu       - pointer to a TMMU structure
    allocator - allocator underlying the MMU to be created
    mmutype   - type of MMU to be created.
        TMMUT_Kernel
            setup a kernel MMU. allocator must be TNULL. the
            newly created MMU will allocate from the kernel.

        TMMUT_Static
            setup a static memory MMU. allocator must point
            to a memheader initialized with TInitMemHead.

        TMMUT_Pooled
            setup a pooled MMU. allocator must point to a
            pooled allocator created with TCreatePool.

        TMMUT_MMU
            setup a MMU on top of another MMU, implementing
            no additional functionality. allocator must point
```

```
                to another MMU.

            TMMUT_Tracking
                setup a tracking MMU on top of another MMU. allocator
                must point to another MMU, or TNULL (which is equivalent
                to a kernel allocator). the resulting MMU will return all
                pending allocations to its parent MMU when it is
                destroyed.

            TMMUT_TaskSafe
                setup a MMU on top of another MMU, implementing safe
                multitasking accesses across tasks, i.e. multiple tasks
                are allowed to operate on the resulting MMU in parallel.
                allocator must point to another MMU, or TNULL (which is
                equivalent to a kernel MMU).

            TMMUT_Message
                setup a special MMU for being used as a message
                allocator. allocator must point to another message MMU,
                or TNULL. aside from special precautions for allocating
                messages, message MMUs also implement multitasking
                safety and tracking capabilities.

            some MMU types may be combined, currently it is possible to
            initialize a MMU implementing TMMUT_TaskSafe|TMMUT_Tracking
            and TMMUT_TaskSafe|TMMUT_Pooled.

        tags      - pointer to an array of tag items

    TAGS
        none defined yet

    RESULTS
        success  - boolean. TFALSE if an invalid combination of a MMU's
                   capabilities was specified.

    NOTES
        a MMU is destroyed with a call to TDestroy.

    SEE ALSO
        TDestroy, TMMUAlloc, TMMUAlloc0, TMMURealloc, TMMUFree,
        TMMUGetSize
```

1.72 TMMUAlloc

```
    NAME
        TMMUAlloc - allocate memory via MMU.

    SYNOPSIS
        mem = TMMUAlloc(mmu,  size)
        TAPtR           TAPTR TUINT

    FUNCTION
        allocate a block of memory via a MMU. returns TNULL when
        the request could not be satisfied.

    INPUTS
        mmu  - pointer to a memory management unit.
        size - size of the allocation [bytes].

    RESULTS
        mem  - pointer to a block of memory, or TNULL.

    SEE ALSO
        TMMUFree, TMMUAlloc0, TMMURealloc, TMMUGetSize,
        TInitMMU
```

1.73 TMMUAlloc0

```
    NAME
        TMMUAlloc0 - allocate blank memory via MMU.

    SYNOPSIS
        mem = TMMUAlloc0(mmu,  size)
        TAPTR            TAPTR TUINT

    FUNCTION
        allocate a blank block of memory via a MMU, i.e. a block
        of memory that is filled with zero-bytes. returns TNULL
        when the request could not be satisfied.

    INPUTS
        mmu  - pointer to a memory management unit.
        size - size of the allocation [bytes].

    RESULTS
        mem  - pointer to a block of memory, or TNULL.

    SEE ALSO
        TMMUAlloc0, TMMUFree, TMMURealloc, TMMUGetSize,
        TInitMMU
```

1.74 TMMUFree

```
    NAME
        TMMUFree - free memory via MMU.

    SYNOPSIS
        TMMUFree(mmu,  mem)
                 TAPTR TAPTR

    FUNCTION
        free a block of memory via MMU.
```

```
INPUTS
    mmu  - pointer to a memory management unit.
    mem  - block of memory to be freed.

SEE ALSO
    TMMUAlloc, TMMUAlloc0, TMMURealloc, TMMUGetSize,
    TInitMMU
```

1.75 TMMURealloc

```
NAME
    TMMURealloc - resize a block of memory via MMU.

SYNOPSIS
    newmem = TMMURealloc(mmu,  oldmem, size)
    TAPTR                TAPTR TAPTR   TUINT32

FUNCTION
    resizes a memory block that was previously allocated from a MMU
    to the specified size, and returns a valid pointer to the resized
    block of memory, or TNULL when the memory block could not be
    resized.

    when a memory block is passed, and the specified size is zero,
    the memory block will be returned to the pool, and the result of
    this function is TNULL.

    when a size is specified, and the memory block is TNULL, this
    function will try to allocate a new block of the given size.

    if mem is TNULL and size is zero, this function will return
    TNULL.

INPUTS
    mmu     - pointer to a memory management unit.
    oldmem  - block of memory to be resized.
    size    - new size of the memory block.

NOTES
    reallocation may require that the given block of memory
    needs to be moved in memory, i.e. pointers to this area
    may become invalid.

RESULTS
    mem  - pointer to a resized (or freshly allocated) block of
           memory, or TNULL.

SEE ALSO
    TMMUAlloc, TMMUAlloc0, TMMUFree, TMMUGetSize,
    TInitMMU
```

1.76 TMMUGetSize

```
NAME
    TMMUGetSize - get size of an allocation from a MMU.

SYNOPSIS
    size = TMMUGetSize(mmu,  mem)
    TUINT              TAPTR TAPTR

FUNCTION
    this function returns the size of an allocation made
    with TMMUAlloc, TMMUAlloc0, or TMMURealloc.

INPUTS
    mem  - previously allocated block of memory, or TNULL.

RESULTS
    size - size of the allocation [bytes].

SEE ALSO
    TMMUAlloc, TMMUAlloc0, TMMURealloc, TMMUFree,
    TInitMMU
```

1.77 TMMUAllocHandle

```
NAME
    TMMUAllocHandle - allocate a handle.

SYNOPSIS
    mem = TMMUAllocHandle(mmu,  destructor,  size)
    TUINT                 TAPTR TDESTROYFUNC TUINT

FUNCTION
    allocate a generic handle with destructor. this function
    expects and initializes a heading THNDL structure in the
    allocated block of memory.

    a handle allocated with this function can be destroyed
    with TDestroy, which will call the supplied destructor
    before the allocated memory is returned to its MMU.

INPUTS
    mmu        - memory manager
    destructor - destructor function being invoked with
                 TDestroy, or TNULL
    size       - total size of the allocation, including
                 the heading THNDL structure

RESULTS
    mem        - handle, or TNULL

SEE ALSO
    TDestroy, TMMUAllocHandle0, TMMUFreeHandle, TInitMMU
```

1.78 TMMUAllocHandle0

```
NAME
    TMMUAllocHandle0 - allocate a handle with blank memory.

SYNOPSIS
    mem = TMMUAllocHandle0(mmu,  destructor,  size)
    TUINT                  TAPTR TDESTROYFUNC TUINT

FUNCTION
    allocate a generic handle with destructor. this function
    expects and initializes a heading THNDL structure in the
    allocated block of memory.

    a handle allocated with this function can be destroyed
    with TDestroy, which will call the supplied destructor
    before the allocated memory is returned to its MMU.

    unlike TMMUAllocHandle, this function will zero out
    the memory followed by the heading THNDL structure.

INPUTS
    mmu        - memory manager
    destructor - destructor function being invoked with
                 TDestroy, or TNULL
    size       - total size of the allocation, including
                 the heading THNDL structure

RESULTS
    mem        - handle, or TNULL

SEE ALSO
    TDestroy, TMMUAllocHandle, TMMUFreeHandle, TInitMMU
```

1.79 TMMUFreeHandle

```
NAME
    TMMUFreeHandle - free a handle

SYNOPSIS
    TMMUFreeHandle(handle)
                   TAPTR

FUNCTION
    free a handle and return its memory to the MMU it was
    allocated from. unlike TDestroy, this function will not
    call a handle's destructor.

INPUTS
```

```
    handle     - handle allocated with TMMUAllocHandle or
                 TMMUAllocHandle0

SEE ALSO
    TMMUAllocHandle, TDestroy, TInitMMU
```

1.80 TDestroy

```
NAME
    TDestroy - destroy a generic handle

SYNOPSIS
    result = TDestroy(handle)
    TINT              TAPTR

FUNCTION
    call a handle's destroy function.

    the destroy function's object-specific return value will
    be returned to the caller.

    if handle is TNULL, this function returns 0.

INPUTS
    handle   - generic handle such as allocated with
               TMMUAllocHandle or TMMUAllocHandle0.
               it is safe to pass TNULL here.

RESULTS
    result   - return value, as returned by the handle's
               destroy function. 0 if handle is TNULL or
               when a handle's destroy function is TNULL.

SEE ALSO
    TMMUAllocHandle, TMMUFreeHandle, TInitMMU
```

1.81 TAddSockPort

```
NAME
    TAddSockPort - make msgport available in internet namespace

SYNOPSIS
    portnr = TAddSockPort(msgport, portnr, tags)
    TUINT                 TPORT*   TUINT   TTAGITEM*

FUNCTION
    add a messageport to the internet namespace and make it
    available on the given internet port number. if portnr is
    zero, this function will try to allocate a new port number
    and bind the messageport to it. in either case, the internet
```

```
        port number will be returned to the caller. a return value
        of zero indicates failure.

    INPUTS
        msgport - messageport to be added to the internet namespace
        portnr  - dedicated internet port number to add the messageport
                  to, or zero for no preference
        tags    - pointer to a list of tag items

    TAGS
        TSock_IdleTimeout, TTIME *
            pointer to a time structure holding a timeout for idle
            internet connections to this messageport. when this timeout
            expires, the respective connection will be dropped without
            further notice, and further communication with that peer
            will be rejected unless the peer reconnects to this
            messageport.
            default: 128 seconds

        TSock_MaxMsgSize, TUINT
            maximum size allowed for a single message incoming via
            network, in bytes. a peer sending larger messages will be
            silently dropped without further notice, and further
            communication with that peer will be rejected unless it
            reconnects to this messageport.
            default: -1 (no limit)

    RESULTS
        portnr  - actual internet port number to which the messageport
                  was added, or zero for failure.

    SEE ALSO
        TFindSockPort, TRemSockPort, TCreatePort
```

1.82 TFindSockPort

```
    NAME
        TFindockPort - find a remote message port.

    SYNOPSIS
        msgport = TFindSockPort(task, ipname, portnr, tags)
        TPORT*                  TAPTR TSTRPTR TUINT16 TTAGITEM*

    FUNCTION
        find a remote messageport that has been announced to
        the internet namespace with TAddSockPort. a proxy
        to the remote messageport will be returned.

    INPUTS
        task    - task, referring to the caller's current context
        ipname  - ip name string
        portnr  - internet port number
        tags    - pointer to a list of tag items
```

```
TAGS
    TSock_ReplyTimeout, TTIME *
        pointer to a time structure holding a timeout for
        replies pending over remote connections. when the
        timeout expires, the message port will fall into a
        'broken' state, and reject further communication with
        the remote peer.
        default: 32 seconds

RESULTS
    msgport - proxy to the remote msgport, or TNULL on failure.

SEE ALSO
    TFindSockPort, TRemSockPort, TCreatePort
```

1.83 TRemSockPort

```
NAME
    TRemSockPort - remove a message port from publicity.

SYNOPSIS
    TRemSockPort(msgport)
                 TPORT*

FUNCTION
    remove a messageport from the internet namespace.

INPUTS
    msgport - msgport that has previously been added to the
              internet namespace with TAddSockPort.

SEE ALSO
    TFindSockPort, TAddSockPort, TCreatePort
```