# The Reference Manual for the int**e**nt$^{TM}$ Kernel

**VERSION 1.48**

**1. INTRODUCTION** ..... **13**

1.1 A Real Time Operating System ..... 13

1.2 Portability ..... 13

**2. TOOL HANDLING** ..... **14**

2.1 Calling Tools ..... 14

2.2 Tool structure ..... 15

2.3 Tool management functions ..... 16

*2.3.1 Decrement the reference count of the specified tool - sys/kn/tool/deref ..... 17*

*2.3.2 Increment the reference count of the specified tool - sys/kn/tool/ref ..... 17*

*2.3.3 Get pointer to an executable tool - sys/kn/tool/open ..... 17*

*2.3.4 Flush all unreferenced tools from memory - sys/kn/tool/flush ..... 17*

*2.3.5 Flush specific tool from memory - sys/kn/tool/flushname ..... 18*

*2.3.6 Copy the name of a tool into the user's buffer - sys/kn/tool/getname ..... 18*

*2.3.7 Run-time stack tracing - sys/kn/tool/stktrace ..... 18*

*2.3.8 Find tool containing specified code address - sys/kn/tool/lookup ..... 18*

*2.3.9 Set stack pointer and jump to specified address, clearing up stack - sys/kn/tool/setspngo ..... 19*

*2.3.10 Return information about tool-list memory usage - sys/kn/tool/memstats ..... 19*

*2.3.11 Return the reference count of a given tool - sys/kn/tool/getrefcount ..... 19*

*2.3.12 Enumerate the tool list - sys/kn/tool/enumerate ..... 19*

2.4 Tool-open failures ..... 19

*2.4.1 Tool-open Failure Callback Function ..... 20*

*2.4.2 This function frees the error information structure passed to a tool-open failure callback handler - sys/kn/tool/dispose_error ..... 20*

*2.4.3 Error information for callback function ..... 21*

*2.4.4 Special tool-open error codes ..... 21*

**3. PROCESS MANAGEMENT** ..... **22**

3.1 Process States ..... 22

3.2 Process Management Function Descriptions ..... 24

*3.2.1 Create a process - sys/kn/proc/create ..... 24*

*3.2.2 Delete a process - sys/kn/proc/delete ..... 25*

*3.2.3 Start a process - sys/kn/proc/start ..... 25*

*3.2.4 Stop the execution of a running process - sys/kn/proc/exit ..... 26*

*3.2.5 Terminate a process - sys/kn/proc/terminate ..... 26*

*3.2.6 Sleep for a specified time - sys/kn/proc/sleep ..... 26*

*3.2.7 Wake a sleeping process - sys/kn/proc/wake* ........ *27*
*3.2.8 Suspend the operation of a process - sys/kn/proc/suspend* ........ *27*
*3.2.9 Decrement the suspend count for a process - sys/kn/proc/resume* ........ *28*
*3.2.10 Yield CPU - sys/kn/proc/deschedule* ........ *28*
*3.2.11 Cause scheduling to take place - sys/kn/proc/ideschedule* ........ *28*
*3.2.12 Alter the values in the process control block - sys/kn/proc/setparams* ........ *29*
*3.2.13 Acquire information about a process - sys/kn/proc/getparams* ........ *29*
*3.2.14 Wait for a child process to stop or terminate, and delete it if it has terminated - sys/kn/proc/wait* ........ *29*
*3.2.15 Wait for a child process to terminate - sys/kn/proc/chld* ........ *30*
*3.2.16 Change the priority of the calling process - sys/kn/proc/chpri* ........ *30*
*3.2.17 Return the priority of the calling process - sys/kn/proc/getpri* ........ *30*
*3.2.18 Set the parent process of the specified process - sys/kn/proc/chppid* ........ *30*
*3.2.19 Sleep until event occurs, storing any spurious wakes - sys/kn/proc/devsleep* ........ *30*
*3.2.20 Add routines to atexit list - sys/kn/proc/atexit* ........ *31*
3.3 PROCESS CREATION HELPER FUNCTIONS ........ 31
*3.3.1 Create and start a process on the same processor as the calling process - sys/kn/proc/exec/local* ........ *31*
*3.3.2 Create and start a process on a specified processor - sys/kn/proc/exec/remote* ........ *31*
*3.3.3 Create and start a process on a processor chosen by the kernel - sys/kn/proc/exec/any* ........ *32*
3.4 SPAWN STRUCTURE FUNCTIONS ........ 32
*3.4.1 Create a spawn structure - sys/kn/proc/spawn/make* ........ *32*
*3.4.2 Create a spawn structure (exec-style) - sys/kn/proc/spawn/emake* ........ *33*
*3.4.3 Modify file descriptor in spawn structure - sys/kn/proc/spawn/modfd* ........ *33*
*3.4.4 Add global data initialisation in spawn structure - sys/kn/proc/spawn/modglobs* ........ *34*
*3.4.5 Add SPAWN_PARENT record to spawn structure - sys/kn/proc/spawn/modparent* ........ *34*
*3.4.6 Add SPAWN_STACK record to spawn structure - sys/kn/proc/spawn/modstack* ........ *34*
*3.4.7 Add SPAWN_SIGMASK record to spawn structure - sys/kn/proc/spawn/modsig* ........ *35*
*3.4.8 Add SPAWN_STACKLIMIT record to spawn structure - sys/kn/proc/spawn/modstklimit* ........ *35*
*3.4.9 Adds requirement to use local PID to spawn structure - sys/kn/proc/spawn/modlocal* ........ *35*
3.5 SPAWN STRUCTURE MACROS ........ 36

**4. PROCESS MANAGEMENT DATA STRUCTURES ........ 38**

4.1 PROCESS ID ........ 38
4.2 HIERARCHICAL PROCESS TABLE ........ 38
4.3 NETWORK-UNIQUE PROCESS IDS ........ 40
4.4 PROCESSOR ID HANDLING FUNCTIONS ........ 40

*4.4.1 Gets the PID of the calling process - sys/kn/proc/pid/get* ....................................................41
*4.4.2 Return an array of all valid PIDs - sys/kn/proc/pid/enumerate*.............................................41

**5. PROCESS SCHEDULING** ..........................................................................................................42

5.1 THE INTENT SCHEDULER ................................................................................................................42
*5.1.1 Contexts* ..........................................................................................................................42
5.2 THE DISPATCHER ..........................................................................................................................42
5.3 SCHEDULING POLICY ....................................................................................................................43
5.4 USING MORE THAN ONE SCHEDULING POLICY WITHIN AN INTENT SYSTEM ..........................................45
*5.4.1 Scheduling Example* ..........................................................................................................45
5.5 DEADLINE FAILURES ......................................................................................................................46

**6. PROCESS SYNCHRONISATION** ..............................................................................................48

6.1 COUNTING SEMAPHORES ...............................................................................................................48
*6.1.1 Initialise a Semaphore - sys/kn/sem/init*...................................................................................49
*6.1.2 Destroy an Unnamed Semaphore - sys/kn/sem/destroy* .....................................................49
*6.1.3 Wait on a Semaphore - sys/kn/sem/wait*.................................................................................49
*6.1.4 Wait on a Semaphore, non blocking - sys/kn/sem/trywait* ...................................................49
*6.1.5 Decrement the semaphore count by the specified amount without blocking - sys/kn/sem/trymwait*..........................................................................................................................49
*6.1.6 Wait on a Semaphore with timeout - sys/kn/sem/timedwait* .................................................50
*6.1.7 Post to a Semaphore - sys/kn/sem/post* ..................................................................................50
*6.1.8 Get the Value of a Semaphore - sys/kn/sem/getvalue*.........................................................50
6.2 MUTEXES.......................................................................................................................................50
*6.2.1 Initialise Mutex - sys/kn/mtx/init*...............................................................................................51
*6.2.2 Destroy Mutex - sys/kn/mtx/destroy*.........................................................................................52
*6.2.3 Lock Mutex - sys/kn/mtx/lock* ...................................................................................................52
*6.2.4 Lock Mutex, non blocking - sys/kn/mtx/trylock*.......................................................................52
*6.2.5 Lock Mutex with timeout - sys/kn/mtx/timedlock* ...................................................................53
*6.2.6 Unlock Mutex - sys/kn/mtx/unlock*...........................................................................................53
*6.2.7 Locks a mutex, retrying if interrupted by a signal - sys/kn/mtx/siglock* ................................53
*6.2.8 Attempt to lock a mutex with timeout, retrying if interrupted by a signal - sys/kn/mtx/sigtimedlock* ..........................................................................................................54
*6.2.9 Return lock status of mutex - sys/kn/mtx/islocked* ................................................................54
6.3 EVENT FLAGS ...............................................................................................................................54
*6.3.1 Initialise an event flag structure - sys/kn/evf/init*....................................................................54
*6.3.2 Destroy an event flag - sys/kn/evf/destroy* ............................................................................54
*6.3.3 Set the flag pattern of an event flag - sys/kn/evf/set*............................................................54

*6.3.4 Clear the flag pattern of an event flag - sys/kn/evf/clr*....54
*6.3.5 Waiting on an Eventflag*....55
*6.3.6 Wait on event flag until a specific condition is fulfilled - sys/kn/evf/wait*....55
*6.3.7 Test event flag for specified event flag pattern, non blocking - sys/kn/evf/trywait*....55
*6.3.8 Wait on event flag for specific condition (blocking), with timeout - sys/kn/evf/timedwait*....56
*6.3.9 Get event flag information - sys/kn/evf/info*....56
6.4 READER/WRITER LOCKS....56
*6.4.1 Initialises a reader/writer lock - sys/kn/rwlock/init*....56
*6.4.2 Destroys a reader/writer lock - sys/kn/rwlock/destroy*....57
*6.4.3 Waits on a reader/writer lock (blocking, no timeout) - sys/kn/rwlock/wait*....57
*6.4.4 Waits on a reader/writer lock (non-blocking) - sys/kn/rwlock/trywait*....57
*6.4.5 Waits on a reader/writer lock (blocking, with timeout) - sys/kn/rwlock/timedwait*....57
*6.4.6 Unlocks a reader/writer lock - sys/kn/rwlock/unlock*....57
*6.4.7 When holding the lock as a reader, become a writer - sys/kn/rwlock/upgrade*....57
6.5 MAILBOXES....58
*6.5.1 Mail Messages*....58
*6.5.2 Allocate a mailbox - sys/kn/mbox/alloc*....59
*6.5.3 Free a mailbox - sys/kn/mbox/free*....59
*6.5.4 Send Message to mailbox - sys/kn/mbox/send*....59
*6.5.5 Read mail from mailbox - sys/kn/mbox/read*....59
*6.5.6 Read mail from a mailbox, non-blocking - sys/kn/mbox/tryread*....60
*6.5.7 Read mail from a mailbox, with blocking and timeout – sys/kn/mbox/timedread*....60
*6.5.8 Set the callback function for the specified mailbox - sys/kn/mbox/setcallback*....60
*6.5.9 Call the specified function with the message list of the mailbox - sys/kn/mbox/enumerate*.61
6.6 SYNCHRONISATION GROUPS....61
*6.6.1 Initialise Synchronisation Group - sys/kn/sgrp/init*....62
*6.6.2 Destroy Synchronisation Group - sys/kn/sgrp/destroy*....62
*6.6.3 Associate Mutex with Synchronisation Group - sys/kn/sgrp/mtx_assoc*....62
*6.6.4 Associate Semaphore with Synchronisation Group - sys/kn/sgrp/sem_assoc*....62
*6.6.5 Associate Mailbox with Synchronisation Group -sys/kn/sgrp/mbox_assoc*....62
*6.6.6 Associate Event Flag with Synchronisation Group - sys/kn/sgrp/evf_assoc*....62
*6.6.7 Associate Synchronisation Group with Synchronisation Group - sys/kn/sgrp/sgp_assoc*....62
*6.6.8 Disassociate Mutex from Synchronisation Group - sys/kn/sgrp/mtx_disassoc*....63
*6.6.9 Disassociate Semaphore from Synchronisation Group - sys/kn/sgrp/sem_disassoc*....63
*6.6.10 Disassociate Mailbox from Synchronisation Group - sys/kn/sgrp/mbox_disassoc*....63
*6.6.11 Disassociate EVF Condition from Synchronisation Group - sys/kn/sgrp/evf_disassoc*....63
*6.6.12 Disassociate Event Flag from Synchronisation Group - sys/kn/sgrp/evf_destroy*....63

*6.6.13 Disassociate Synchronisation Group from Synchronisation Group - sys/kn/sgrp/sgp_disassoc* ....... *63*
*6.6.14 Wait on Synchronisation Group – sys/kn/sgrp/wait* ....... *63*
*6.6.15 Test Synchronisation Group, non-blocking – sys/kn/sgrp/trywait* ....... *64*
*6.6.16 Wait on Synchronisation Group, with timeout – sys/kn/sgrp/timedwait* ....... *64*
6.7 STRUCTURE OF SYNCHRONISATION GROUP INFORMATION RECORDS ....... 64

**7. MEMORY MANAGEMENT ....... 66**

7.1 DYNAMIC MEMORY ALLOCATION ....... 67
*7.1.1 Allocate memory from specified type of memory object - sys/kn/mem/alloc<type>* ....... *67*
*7.1.2 Allocates memory with a specified alignment - sys/kn/mem/allocaligned* ....... *68*
*7.1.3 Allocate memory from specified memory object - sys/kn/mem/alloc* ....... *68*
*7.1.4 Free memory allocated from corresponding allocation tool - sys/kn/mem/free* ....... *69*
*7.1.5 Re-allocate memory - sys/kn/mem/realloc* ....... *69*
*7.1.6 Check structure of all system memory objects - sys/kn/mem/check* ....... *69*
*7.1.7 Get pointer to named memory object - sys/kn/mem/lookup* ....... *69*
*7.1.8 Return size of block of memory - sys/kn/mem/size* ....... *70*
*7.1.9 Returns the memory object associated with a block - sys/kn/mem/getobj* ....... *70*
7.2 VIRTUAL MEMORY SERVICES ....... 70
*7.2.1 Lock Memory - sys/kn/mem/lock* ....... *70*
*7.2.2 Unlock Memory - sys/kn/mem/unlock* ....... *71*
7.3 MEMORY OBJECT METHODS ....... 71
*7.3.1 Constructor tool - <class name>/_new* ....... *71*
*7.3.2 Initialise memory object - _init* ....... *72*
*7.3.3 Allocate memory from memory object - alloc (xmethod)* ....... *72*
*7.3.4 Free memory to memory object - free (xmethod)* ....... *72*
*7.3.5 Add memory block to memory object - addblock* ....... *72*
*7.3.6 Return statistics about memory usage - info* ....... *72*
*7.3.7 Check memory object structure - check* ....... *72*
*7.3.8 Return size of largest available memory block - largest* ....... *72*
7.4 MEMORY FLUSHING ....... 72
*7.4.1 Add callback routine to memory flush list - sys/kn/mem/addflush* ....... *73*
*7.4.2 Remove a callback function from the memory flush list - sys/kn/mem/removeflush* ....... *73*
*7.4.3 Process memory flush callback list - sys/kn/mem/flush* ....... *73*

**8. TIMER MANAGEMENT ....... 75**

*8.1.1 Set up a timer, using the priority of the calling process - sys/kn/timer/set* ....... *76*
*8.1.2 Set up a timer, using the priority specified in the timer data structure - sys/kn/timer/dset* ....... *77*

*8.1.3 Unset timer - sys/kn/timer/unset ... 77*
*8.1.4 Unsets periodic timer from within its own handler - sys/kn/timer/handler_unset ... 77*

**9. INTERRUPT HANDLING ... 78**

9.1 RESTRICTIONS ON INTERRUPT HANDLERS ... 78

**10. EXCEPTION HANDLING ... 80**

10.1 EXCEPTION NUMBERS ... 81
10.2 THE PROCESS/DEBUGGER THIRD LEVEL EXCEPTION HANDLER ... 82
*10.2.1 Enables a process to throw a software exception - process_exception_handler ... 82*
10.3 EXCEPTION HANDLING FUNCTIONS ... 83
*10.3.1 Register a process's exception handler with the kernel - sys/kn/exc/set ... 83*
*10.3.2 Deregister a process's exception handler with the kernel - sys/kn/exc/unset ... 83*
*10.3.3 Register a system-wide exception handler with the kernel - sys/kn/exc/setsys ... 83*
*10.3.4 Deregister a system-wide exception handler with the kernel - sys/kn/exc/unsetsys ... 83*
*10.3.5 Register a debugger's exception handler with the kernel - sys/kn/exc/setdbg ... 83*
*10.3.6 Deregister a debugger's exception handler with the kernel - sys/kn/exc/unsetdbg ... 84*
*10.3.7 Throw a software exception - sys/kn/exc/throw ... 84*

**11. SIGNALS ... 85**

11.1 GENERATING SIGNALS ... 86
11.2 SIGNAL ACTIONS ... 86
11.3 DELIVERING A SIGNAL TO A PROCESS ... 87
11.4 DISABLING SIGNAL DELIVERY ... 87
11.5 SIGNAL FUNCTIONS ... 88
*11.5.1 Send a signal to a process- sys/kn/sig/kill ... 88*
*11.5.2 Examine or change signal action - sys/kn/sig/action ... 88*
*11.5.3 Examine or change blocked signals - sys/kn/sig/procmask ... 88*
*11.5.4 Examine pending signals - sys/kn/sig/pending ... 89*
*11.5.5 Return set of signals which have been raised but not yet taken - sys/kn/sig/raised ... 89*
*11.5.6 Wait for a signal - sys/kn/sig/suspend ... 89*
*11.5.7 Disable or enable signal handling - sys/kn/sig/setflag ... 89*
*11.5.8 Sets handler for specified signal - sys/kn/sig/signal ... 90*
11.6 SIGNAL SET FUNCTIONS ... 90
*11.6.1 Creates an empty set - sys/kn/sig/emptyset ... 90*
*11.6.2 Creates a full set - sys/kn/sig/fillset ... 90*
*11.6.3 Adds a signal to a set - sys/kn/sig/addset ... 90*
*11.6.4 Delete a signal from the set - sys/kn/sig/delset ... 90*

*11.6.5 Tests set to see if signal is a member - sys/kn/sig/ismember............................................91*

*11.6.6 Modifies the set of pending signals - sys/kn/sig/setpending...............................................91*

*11.6.7 Modifies the set of pending signals - sys/kn/sig/orpending................................................91*

**12. EVENT TOOLS ........................................................................................................................92**

*12.1.1 Initialises an event tracker - sys/kn/event/init.....................................................................92*

*12.1.2 Destroys an event tracker - sys/kn/event/destroy..............................................................93*

*12.1.3 Waits on an event tracker (blocking, no timeout) - sys/kn/event/wait.................................93*

*12.1.4 Waits on an event tracker (non-blocking) - sys/kn/event/trywait........................................94*

*12.1.5 Waits on an Event tracker (blocking, with timeout) - sys/kn/event/timedwait.....................94*

*12.1.6 Alters the state of an event tracker - sys/kn/event/alter......................................................94*

*12.1.7 Gets the value of an event tracker - sys/kn/event/info........................................................94*

*12.1.8 Waits on an event tracker in a specified way (blocking, no timeout) - sys/kn/event/wait_fn ..............................................................................................................................................94*

*12.1.9 Waits on an event tracker in a specified way (non-blocking) - sys/kn/event/trywait_fn......94*

*12.1.10 Waits on an event tracker in a specified way (blocking, with timeout) - sys/kn/event/timedwait_fn.........................................................................................................94*

*12.1.11 Alters the state of an event tracker in a specified way - sys/kn/event/alter_fn.................95*

*12.1.12 Call a function for each caller waiting on an event tracker - sys/kn/event/enumerate......95*

**13. CALLBACK MANIPULATION FUNCTIONS...............................................................................96**

13.1 THE CALLBACK DATA STRUCTURE .......................................................................................96

13.2 THE CALLBACK HANDLER FUNCTION....................................................................................97

13.3 KERNEL CALLBACK MANIPULATION FUNCTIONS ...................................................................97

*13.3.1 Set a callback - sys/kn/callback/set ......................................................................................97*

*13.3.2 Unset a callback - sys/kn/callback/unset ..............................................................................97*

*13.3.3 Disable/Enable callbacks for the calling process - sys/kn/callback/setflag.........................97*

*13.3.4 Process any pending callbacks - sys/kn/callback/process .................................................98*

*13.3.5 Returns the value of the PF_CALLBACK_OCCURRED flag for the calling process - sys/kn/callback/occurred......................................................................................................98*

*13.3.6 Clears the PF_CALLBACK_OCCURRED flag for the calling process - sys/kn/callback/clr_occurred ..................................................................................................98*

*13.3.7 Indicate whether a callback is currently pending for the calling process - sys/kn/callback/pending .........................................................................................................98*

**14. NAMED DATA AREA FUNCTIONS ...........................................................................................99**

*14.1.1 Associate the specified pointer with the specified string - sys/kn/nda/name......................99*

*14.1.2 Delete the NDA record for the specified string - sys/kn/nda/del .........................................99*

*14.1.3 Look up the NDA record for the specified string - sys/kn/nda/find......99*

**15. ATOMS......100**

15.1 ATOM VALUES......100
15.2 STATIC ATOMS......100
15.3 DYNAMIC ATOMS......100
15.4 ATOM FUNCTIONS......100
*15.4.1 Return atom value for specified string, create new atom if required - sys/kn/atom/add...100*
*15.4.2 Dereference the specified atom, delete if unreferenced - sys/kn/atom/del......101*
*15.4.3 Return the atom value corresponding to the specified string - sys/kn/atom/find......101*
*15.4.4 Return a copy of the string corresponding to the specified atom - sys/kn/atom/getname 101*
*15.4.5 Increment the reference count of an atom - sys/kn/atom/ref......101*

**16. KERNEL NOTIFICATION FUNCTIONS......102**

16.1 SYSTEM EVENTS......102
16.2 THE EVENT HANDLING FUNCTION......102
16.3 KERNEL NOTIFICATION FUNCTIONS......103
*16.3.1 Announce that an event has taken place - sys/kn/notify/announce......103*
*16.3.2 Register an interest in being notified when a particular event occurs - sys/kn/notify/subscribe......103*
*16.3.3 Register an interest in being notified when a particular named event occurs - sys/kn/notify/subscribe_name......103*
*16.3.4 Withdraw previous subscription to be notified when a certain event is generated - sys/kn/notify/unsubscribe......104*
*16.3.5 Register as a generator of a particular kind of event - sys/kn/notify/generator......104*
*16.3.6 Register as a generator of a particular kind of named event - sys/kn/notify/generator_name......104*
*16.3.7 Cease to be a generator of a particular type of event - sys/kn/notify/ungenerator......104*

**17. ATOMIC LIST FUNCTIONS......105**

*17.1.1 Add a node to the head of the list - sys/kn/atomic/addhead......105*
*17.1.2 Add a node to the tail of the list - sys/kn/atomic/addtail......105*
*17.1.3 Add a node after the specified list node - sys/kn/atomic/addnode......105*
*17.1.4 Add a node before the specified list node - sys/kn/atomic/addnodeb......105*
*17.1.5 Remove the node from the head of the specified list - sys/kn/atomic/removehead......106*
*17.1.6 Remove the specified node from its list - sys/kn/atomic/removenode......106*
*17.1.7 Moves the entire contents of one list onto another - sys/kn/atomic/movelist......106*

**18. MINI ATOMIC BLOCKS......107**

*18.1.1 Enter a scheduler MAB - sys/kn/mab/begin_s ........ 107*
*18.1.2 Leave a scheduler MAB - sys/kn/mab/end_s ........ 107*
*18.1.3 Enter a thread MAB - sys/kn/mab/begin_t ........ 107*
*18.1.4 Leave a thread MAB - sys/kn/mab/end_t ........ 107*
*18.1.5 Enter an interrupt MAB - sys/kn/mab/begin_i ........ 107*
*18.1.6 Leave an interrupt MAB - sys/kn/mab/end_i ........ 108*

**19. AVL TREE MANAGEMENT FUNCTIONS ........ 109**

19.1 KERNEL MANAGEMENT FUNCTIONS FOR THE AVL TREE ........ 110
*19.1.1 Initialises an AVL tree header - sys/kn/avl/init ........ 110*
*19.1.2 Initialises an AVL tree header with relocatable stack - sys/kn/avl/init_rs ........ 110*
*19.1.3 Initialises an AVL tree header to permit elements with duplicate keys - sys/kn/avl/init_dup ........ 111*
*19.1.4 Initialises an AVL tree header with relocatable stack to permit elements with duplicate keys - sys/kn/avl/init_dup_rs ........ 111*
*19.1.5 Switch the location of the relocatable stack - sys/kn/avl/move_rs ........ 111*
*19.1.6 Deinitialises an AVL tree header - sys/kn/avl/deinit ........ 111*
*19.1.7 Inserts a new node into an AVL tree - sys/kn/avl/insert ........ 111*
*19.1.8 Removes a node from an AVL tree, rebalancing the tree - sys/kn/avl/remove ........ 112*
*19.1.9 Removes all the elements from an AVL tree - sys/kn/avl/removeall ........ 112*
*19.1.10 Removes elements within the given range from an AVL tree - sys/kn/avl/removerange 112*
*19.1.11 Finds an element within a tree - sys/kn/avl/find ........ 112*
*19.1.12 Finds a node with a specified key value within a tree - sys/kn/avl/findkey ........ 113*
*19.1.13 Returns the maximal element within an AVL tree - sys/kn/avl/maximum ........ 113*
*19.1.14 Returns the minimal element within an AVL tree - sys/kn/avl/minimum ........ 113*
*19.1.15 Enumerates all elements in an AVL tree, calling a function for each one - sys/kn/avl/enumerate ........ 113*
*19.1.16 Returns the number of elements in an AVL tree - sys/kn/avl/size ........ 113*
*19.1.17 Checks the consistency of an AVL tree - sys/kn/avl/check ........ 113*
*19.1.18 Finds the leftmost element greater than or equal to a key - sys/kn/avl/ubound ........ 113*
*19.1.19 Finds the rightmost element less than or equal to a key - sys/kn/avl/lbound ........ 114*
*19.1.20 Finds the next element left from a given element (ie. next smaller key) - sys/kn/avl/walkleft ........ 114*
*19.1.21 Finds the next element right from a given element (ie. next larger key) - sys/kn/avl/walkright ........ 114*

**20. KERNEL DEVICE FUNCTIONS ........ 115**

*20.1.1 Look up a device in the system mount table - sys/kn/dev/lookup ........ 115*

*20.1.2 Look up a device in the system mount table - sys/kn/dev/rlookup....116*
*20.1.3 Add a device to the system mount table - sys/kn/dev/mount....116*
*20.1.4 Remove the specified device from the system mount table - sys/kn/dev/unmount....116*
*20.1.5 Adds a delayed-mount record for a device to the mount table - sys/kn/dev/mount_delayed ....116*
*20.1.6 Adds a new device driver to a running system - sys/kn/dev/start....116*
*20.1.7 Unmounts and stops a device - sys/kn/dev/stop....117*

**21. STATIC AREAS SUPPORT....118**

21.1 STATIC DATA CACHES....118
21.2 ACCESSING STATICS....118
*21.2.1 Find the data associated with a cache, or allocate it if necessary - sys/kn/statics/statics_get....120*
*21.2.2 Delete the statics associated with the key and clear the cache - sys/kn/statics/statics_delete....121*
*21.2.3 Allocates a static area (used when WDA_SETUP_LENGTH is -1)- alloc....121*
*21.2.4 Performs initialisation on an automatically allocated static area- new....122*
*21.2.5 If setup length is 0 or more then the space is automatically allocated - init....122*
*21.2.6 Wait for statics area to be initialised - wait....122*
*21.2.7 Deinitialise the statics area - deinit....123*
*21.2.8 Deletes the statics area by freeing it - delete....123*

**22. KERNEL ENTROPY COLLECTOR....124**

22.1 ENTROPY COLLECTION OVERVIEW....124
22.2 THE KERNEL ENTROPY COLLECTOR....126
*22.2.1 Entropy Collector Input....126*
*22.2.2 Entropy Collector Output....127*
22.3 KERNEL ENTROPY COLLECTOR FUNCTIONS....127
*22.3.1 Add entropy to the kernel collector pool - sys/kn/entropy/add....127*
*22.3.2 Add timer entropy to the pool - sys/kn/entropy/add_time....127*
*22.3.3 Register the timer clock resolution - sys/kn/entropy/reg_time....127*
*22.3.4 Extract entropy from collector pool - sys/kn/entropy/get....128*
*22.3.5 Extract entropy from collector pool (alternative interface) - sys/kn/entropy/getrand....128*
*22.3.6 Extract entropy asynchronously - sys/kn/entropy/get_async....129*
*22.3.7 Cancel asynchronous entropy request - sys/kn/entropy/get_abort....129*

**23. KERNEL TIME FUNCTIONS....130**

*23.1.1 Get the kernel time - sys/kn/time/get....130*

**24. DATA STRUCTURE DEFINITIONS....131**

24.1 PROCESS CONTROL BLOCK STRUCTURE....131
24.2 SPAWN STRUCTURE....132
24.3 EVENT FLAG INFORMATION STRUCTURE....133
24.4 MAIL MESSAGE HEADER....133
24.5 MEMORY FLUSH LIST NODES....134
24.6 CALLBACK DATA STRUCTURE....134
24.7 NOTIFY DATA STRUCTURE....135
24.8 AVL TREE....135
24.9 TIMER STRUCTURE....136
24.10 ENTROPY COLLECTOR CONTROL STRUCTURE....137

**25. GLOSSARY....139**

# 1. Introduction

The kernel for the intent™ environment is a collection of functions which provide basic facilities such as the transportation of messages, memory management, process distribution, code and data caching, and process scheduling. intent kernel functions are effectively globally accessible tools.

The intent kernel is a microkernel, a set of compact modules which may be easily configured, and which demands only a small memory footprint. Many functions that the kernel of a standard operating system might be expected to contain are not considered to be part of the intent kernel, but instead are categorised separately. Device drivers, for example, are not part of the kernel.

This document provides a detailed overview of the functionality offered by the kernel. Readers intending to perform programming involving the kernel are advised to use this manual in conjunction with the relevant API documents. Each of these documents may be found in the relevant subdirectory of the *sys/kn* directory.

## 1.1 A Real Time Operating System

intent is an environment with real time capabilities. This means that within the system, operations can be guaranteed to take place in a deterministic fashion. This means that given sufficient information about the state of the system, the length of time taken by an operation can be calculated.

The kernel contains functions that govern the system's scheduler, timer and synchronisation mechanisms. It also provides a means of manipulating tools and processes, and handling memory and data structures.

## 1.2 Portability

All functions contained within the intent kernel are portable.

Where the kernel is called upon to use system functionality that is processor-dependent, it is achieved through the CPU Isolation Interface (CII). The CII acts as an interface between the kernel and the processor, and aids in the execution of tasks such as those involving stack allocation, manipulation of the native instruction set, scheduler-related functions and cache-flushing.

The kernel achieves interface with underlying hardware, or software in the case of a hosted system, through use of the Platform Isolation Layer (PIL). The PIL is mainly comprised of the device drivers, and the Platform Isolation Interface (PII). The PII is used by the kernel to effect tasks such as interrupt handling, system startup and shutdown, and exception management. When the system is running hosted on another OS, the PII can also handle memory management, but this is rarely used. Usually, the kernel itself will be responsible for memory management.

Details of PII and CII functions may be found in the relevant section of *The System Programmer's Guide*, and in a number of documents included in the build, such as *sys/cii/api.html* and *sys/pii/api.html*.

# 2. Tool Handling

intent may be regarded as using a 'tool-based' system. Within intent tools are structures in memory which contain code, in the form of subroutines and functions, and data.

In storage, tools exist as non-executable templates. Most tools are stored in VP, and translated into native code at either load time or sysgen time. A tool will always contain only native code when in executable state. Tools may also be stored in native code. This permits some processor-specific operations and optimisations to be incorporated into an intent system if required.

Tools may be bound into an application in two different ways, through static binding or through dynamic binding.

In the case of static binding, intent makes use of an image generation facility. Two of these are available, the Sysbuild Utility and the System Image Generation Utility (Sysgen). The Sysgen utility creates a bootable image from a list of instructions. These describe the tools to be included in the image, the processes and device objects to be created at initialisation, and the system boot sequence. The image may then be booted and executed as an application. The Sysbuild utility receives an applications ‘sysbuild file,’ which contains details of tools and data files which the application requires. Sysbuild then calls upon the underlying Sysgen functionality to create a bootable application image.

In the case of dynamic binding, tools are loaded as they are referenced. This usually takes place when processes are created; the ‘main’ tool, which the process will begin by executing, and any tools *qcall*ed by it, will be dynamically loaded, translated, and bound. Tools may also be referenced as a result of virtual *qcalls* by other tools. Thus tools are brought into memory as and when they are required. This binding process is transparent to the user.

It is usual to use only static binding in embedded systems. Other systems will often combine the two tool loading mechanisms. While the initial system image will customarily be generated using static binding, other tools will often be loaded using dynamic binding, once the system is running.

## 2.1 Calling Tools

Within intent, tools may call one another in a similar manner to that in which one might call a library function in a high-level language such as C.

intent tools make use of three *qcall*s (quick call). These are the 'normal' *qcall*, the virtual *qcall* and the virtual+fixup *qcall*. These calls vary only in their binding semantics, and their differences will not be apparent to the user.

- **"Normal" *qcall***
  The referenced tool is found at bind-time (loaded, translated if necessary and bound if it does not exist in memory), and the calling tool is fixed up to call directly to its entry point. There is no run-time performance penalty for using this method - it is as fast as a subroutine call. The referenced tool remains in memory until the calling tool is removed.

- **"Virtual" *qcall***
  At bind-time, the calling tool is bound to a kernel function with the name of the tool to be called. At run-time, the kernel function finds the referenced tool (loading, translating if necessary and binding it if it does not exist in memory), and then calls it. The referenced tool is dereferenced when the call returns.

This method is considerably slower than a normal *qcall*, but it has the advantage that the tool being called is only referenced during the period of its execution. Therefore, it can be removed from memory if necessary while it is not executing. If this occurs, it is automatically reloaded on the next call to it (and re-translated and re-bound).

- **"Virtual+fixup" *qcall***
  At bind-time, the calling tool is bound to a kernel function with the name of the tool to be called. At run-time, the first time that the call is executed, the kernel function finds the referenced tool (loading, translating if necessary and binding it if it does not exist in memory), and then modifies the calling instruction to point directly to the referenced tool. It then calls the referenced tool. The referenced tool returns directly to the calling tool.

  Subsequent executions of the call are almost as fast as a normal *qcall*, having the overhead of a single constant load instruction. This mechanism is almost as slow as a virtual *qcall* on its first execution, but almost as fast as a normal *qcall* on all subsequent executions. The referenced tool cannot be removed from memory after it is loaded, until the calling tool itself is removed (as with a normal *qcall*).

  This method is commonly used where a number of tools may be called, and it is unknown at assembly-time which one(s) will be used. However, once it is decided which tools are to be used (according to some run-time parameter), the calls to the tools must be as fast as possible. This avoids the memory overhead of loading all of the referenced tools, at the expense of a slower first call.

In general, tools are loaded and fixed up automatically by the kernel binder when they are implicitly referenced by another tool, or by the creation of a process. However, it is possible to open a specified tool manually. This uses the same mechanism as the intent dynamic binder, but can be valuable in situations where, for example, the name of the tool to be called is not known at assembly-time.

## 2.2 Tool structure

In intent, a tool is a structure containing a small header, executable code, and some information for use by the dynamic binder. Below a typical tool is shown, in template format, as generated by translators, assemblers or compilers, and in ready, run-time format, after being processed by the kernel binder.

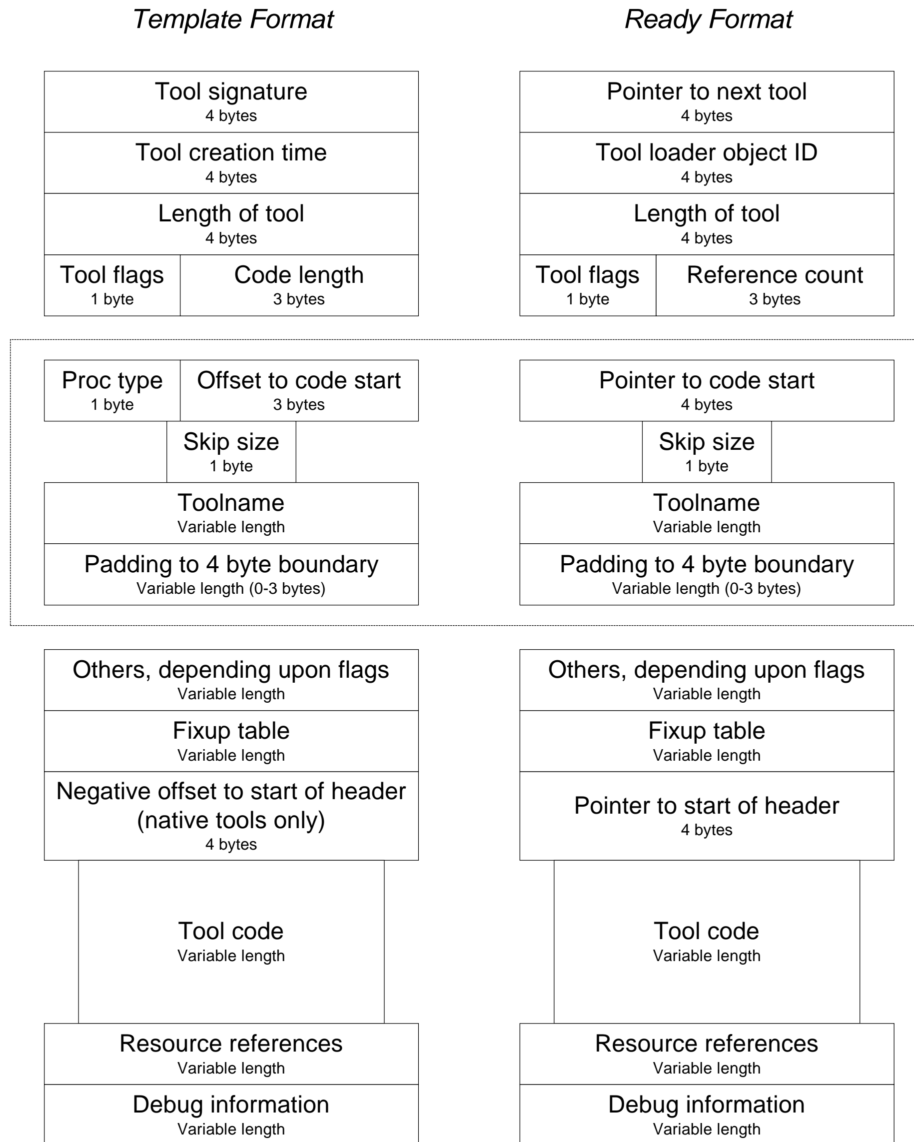


**Diagram 1: Structure of an intent Tool**

One part of the structure (enclosed within the dashed box in each diagram) is repeated one or more times to allow multiple named entry points into the tool. In all except the first, the processor type field is set to zero.

The intent tool header fields are described in detail in *'The Elate Tool Programming Guide'*.

## 2.3 Tool management functions

The functions described in this document allow some control over the behaviour of the kernel's tool handling system.

### 2.3.1 Decrement the reference count of the specified tool - sys/kn/tool/deref

This function decrements the reference count of the specified tool. The reference count indicates the number of tools in memory that are referencing, or using, the specified tool. While the count remains above zero, the tool is retained in memory. If the count reaches zero, this indicates that the tool is no longer required, and it may be flushed out of memory.

The *sys/kn/tool/deref* function may be used to decrement the reference count 'artificially,' so as to allow a particular tool to be flushed from memory.

See Also: *sys/kn/tool/ref*, *sys/kn/tool/open*

### 2.3.2 Increment the reference count of the specified tool - sys/kn/tool/ref

This function increments the reference count of the specified tool. The *sys/kn/tool/ref* function may be used to increase the reference count 'artificially,' so that a tool is not flushed out of memory even when it is being referenced by no other tools.

This functionality is valuable where it is undesirable for specific tools to be flushed, even when the system memory is running low. Device drivers, for example, often need to remain in memory, even when they are not being referenced by any other tool.

See Also: *sys/kn/tool/deref*, *sys/kn/tool/open*

### 2.3.3 Get pointer to an executable tool - sys/kn/tool/open

This function searches for the specified tool. The tool is sought first in the kernel's tool list on the caller's processor. The name of the tool found must be an exact match. If the search is successful, the function returns pointers to the tool. If the tool is not found here, the specified tool loader is called. If the value -1 is specified as the tool loader ID, however, the system will search the list of tools already in memory but will not attempt to load the tool if it is not found.

The tool loader is system-dependent, but on a desktop-type system, it typically attempts to load the file from disk using a set of tool extensions specified at sysgen time to determine the search order. Native tools are loaded in preference to VP tools. Several native code extensions can be specified, and these will be searched in the specified order.

Thus, the tool loader will usually attempt first to find a version of the tool in code native to the processor upon which the system running. It will then seek versions in the code for machines of the same processor family, searching for the best match first. If none such are found, it will resort to the version in VP. The advantage of selecting the native version where possible, is that time may be saved through omission of the translation stage.

If a tool is opened, it is automatically referenced. An open tool, therefore, will remain in memory until it is explicitly dereferenced.

If the specified tool is not found or if another error occurs, both returned pointers are NULL.

See Also: *sys/kn/tool/deref*

### 2.3.4 Flush all unreferenced tools from memory - sys/kn/tool/flush

This function may be used to flush all unreferenced tools from memory when system resources are running low.

See Also: *sys/kn/tool/flushname*

### 2.3.5 Flush specific tool from memory - sys/kn/tool/flushname

This function allows a specific tool to be flushed from memory. If the specified tool has a reference count of zero, this function will simply remove it from memory. If its reference count is non-zero, then there are other tools referencing it, and it cannot yet be removed from memory. In this case, the toolname is 'annulled,' meaning that the tool is flagged as having been flushed. Subsequent attempts to bind to the annulled tool will fail, and the tool must be re-loaded the next time it is opened. Any processes or tools which are using the specified tool when it is annulled will continue to use it until they have finished.

If the toolname is annulled, the tool list is searched for other tools which reference the specified tool, and all tools that reference those tools, etc. If found, their names are also annulled. This prevents any future bind operations from indirectly referencing the flushed tool.

It should be noted that the kernel is never guaranteed to flush a tool immediately and in all cases it may elect to simply mark the tool as flushed as described above, then perform the actual flush at some later point.

See Also: *sys/kn/tool/flush*

### 2.3.6 Copy the name of a tool into the user's buffer - sys/kn/tool/getname

This function is passed a pointer to a particular tool, and copies the textual name of this tool into the specified buffer.

If the specified buffer pointer is NULL, the string cannot be copied. The length of the buffer required, including the null character used to mark the end of the string, is returned. This is equivalent to the length of the string to be copied.

### 2.3.7 Run-time stack tracing - sys/kn/tool/stktrace

This function allows the system to make an accurate stack trace of Java™ methods when Java™ exceptions are thrown. This tool makes use of the translator, which outputs stack trace information inside tools for which the stack trace feature is enabled. Enough information is output to ensure that at every point where a Java™ exception may be thrown, it is possible to perform a stack trace through an instance of that subroutine.

A stack trace can usually be started by calling this subroutine, and passing it the parameter pointer and the link pointer, as the first and second pointer input values respectively.

If a CPU exception has caused a signal, the PII needs to supply the values of the program counter and stack pointer, at the point when the exception was thrown. The PII will have saved these values in its first level exception handler.

On output, the tool pointer may be non zero when the other return values are 0. This occurs when the supplied program counter is in a valid tool, but that tool has no stack trace information.

### 2.3.8 Find tool containing specified code address - sys/kn/tool/lookup

This function returns the header and entrypoint addresses of the tool containing the specified address. If no tool contains this address, NULL pointers are returned. The name of the tool returned can be obtained by calling *sys/kn/tool/getname*.

This function is usually required during debugging. While it is often easy to establish the code address at which an error has occurred, it is more useful to know the identity of the tool that contains this address.

Since this tool returns two values, it cannot currently be directly called from C.

See Also: *sys/kn/tool/getname*

### 2.3.9 Set stack pointer and jump to specified address, clearing up stack - sys/kn/tool/setspngo

This tool is used to jump back to an earlier (or higher) program state. It sets the stack pointer to the specified value, removing any linked stack blocks which become unused by this process, and also dereferencing any virtually called tools whose returns are jumped past.

The jump must be made to a sync instruction. The global pointer is not modified by this call. The parameter pointer and the link pointer are restored to the states appropriate for the point to which execution is jumping. The contents of all other VP registers, however, are undefined after the jump, unless those registers were the subject of a syncreg instruction at the time that the execution originally left that function. In such cases, they will recover the values they held at that time.

There are typically two ways in which the stack pointer and program counter values are determined. The first is to simply store the current value of stack pointer during some function, and store the address of a label at the same stack level (i.e. a label within the same ent block and without any intervening als statements).

The second way is that within a function, one can store the values of the parameter pointer and link pointer registers, and later pass them as parameters to *sys/kn/tool/setspngo.* However, it should be noted that, in this second case, the function in which parameter pointer and link pointer are stored must take a single pointer as input and return a single integer.

### 2.3.10 Return information about tool-list memory usage - sys/kn/tool/memstats

This tool fills in the specified structure with information about the contents of the current tool-list. At present the ELATE_TLINFO structure contains the following fields:

- *totalmem*
  This field contains the total amount of memory used by the tools on the tool-list, in bytes.
- *numtools*
  This field contains the number of tools on the tool-list.

### 2.3.11 Return the reference count of a given tool - sys/kn/tool/getrefcount

*sys/kn/tool/getrefcount* returns the value of the reference counter of the tool specified.

### 2.3.12 Enumerate the tool list - sys/kn/tool/enumerate

This tool enumerates all elements on the tool list. For each tool, the user-defined handling function is called. If the handling function returns a non-zero value, then the enumeration is aborted and the same return value is returned by this tool.

## 2.4 Tool-open failures

A call to *sys/kn/tool/open* may fail for a number of possible reasons. This may be caused by the failure of functions called during the operation of *sys/kn/tool/open*, such as mutex locking or memory

allocation functions. Tool-open failure may also be caused by errors more specific to the tool loading mechanism, such as a range of bind failures, translation failures and loading failures.

Since the tool-open errors are entirely internal to the tool-loading mechanism, they are not assigned system-wide error codes. This prevents the wasteful use of the general error codes space for a set of errors applicable only to a narrow and specific set of circumstances.

Thus, a separate set of error codes have been defined for tool-open errors. They are all positive, and thus can be distinguished from the more generic error codes, which are uniformly negative. The value 0 is usually used to mean ‘no error’ within both sets of error codes.

### 2.4.1 Tool-open Failure Callback Function

When a tool operation fails, a call is made to any user-defined callback function indicated by the process global data field PROC_TFHANDLER. If this field contains NULL, no action is taken, and NULL is returned from *sys/kn/tool/open*. (It should be noted that although this function can be considered to be a callback it is not of the format used by the *sys/kn/callback* functions.)

Error values passed into this function may be either normal intent error codes, or one of the values designed specifically for tool-open failures. The VP macros *iferrno* and *boolerrno* may be used to determine which of these types of error code has been passed to the function, as they will not consider numbers specific to tool-open failures to be error codes.

Error codes simply identify the system condition that has prevented the completion of a certain operation, such as lack of memory, and thus cannot generally be associated with the failure of a specific tool. For this reason, if the callback function is passed a valid error code, the error information pointer will be set to point to the name of the top-level requested tool.

If the callback function is passed one of the special error codes defined below, the contents of the memory indicated by the error information pointer parameter depends on the exact error code.

The callback function can do many things, including printing out the error details, or recording the error information in some way. The behaviours of such callback functions may be divided between those that return to the calling function and those that do not.

A tool-open failure callback function which returns to the caller does not need to take any special action, and can simply process the error information as it desires and then return to the caller which will perform any necessary clean-up actions.

A tool-open failure callback function which does not return to the caller (such as a handler which throws a high-level language exception or performs a C-style longjmp) must call the function *sys/kn/tool/dispose_error* to ensure that the error information is cleared up properly.

### 2.4.2 This function frees the error information structure passed to a tool-open failure callback handler - sys/kn/tool/dispose_error

This function should only be called if the handler function will not return to its caller.

This function takes two parameters, a pointer to the error information, and an error code. Both parameters to this function should be passed through from the tool-open failure handler function unmodified.

### 2.4.3 Error information for callback function

The pointer passed to the callback function indicates an area of error information including a 'tool binding list.' This is a structure that contains information concerning the error that has occurred, and takes the format described below. All strings within this structure are null-terminated.

| | |
|---|---|
| String | This is the name of the tool containing the error. |
| String(s) | This part of the structure is optional, and may contain multiple strings. Each of these strings is the name of the tool referencing the previous tool to be named. |
| String | This is the name of the tool requested at the top level by a call to *sys/kn/tool/open*. The tool named here is a 'primary tool.' |
| Byte | 0 |

This structure describes the hierarchy of tools being bound when the error was found. The second part of the structure may vary radically in size, but the structure can be decoded by treating the first string as the name of the tool containing the error, and the last string as that of the tool requested at top level. If there is only one string in the structure, the error has been found in the top-level tool.

### 2.4.4 Special tool-open error codes

The following error codes are defined. The error information for the callback function varies depending upon the error code. Below, for each error code, the corresponding error information is described.

| Code | Error | Error information |
|---|---|---|
| 1 | Bad resource number | 4-byte integer specifying the invalid resource number, followed by a "tool-binding list." |
| 2 | NDA not found | The name string of the named data area which was not found, followed by a "tool-binding list." |
| 3 | Failed to create atom | The string for which an atom could not be created, followed by a "tool-binding list." |
| 4 | Failed to find resource | Resource name which could not be found, followed by a "tool-binding list." |
| 5 | PII opentool failed | Name of failing tool, followed by a "tool-binding list." |
| 6 | Tool not found | Name of tool which was not found, followed by a "tool-binding list." |
| 7 | I/O error | Name of tool which could not be read, followed by a "tool-binding list." |
| 8 | Translator not found | Name of translator which was not found, followed by a "tool-binding list." |
| 9 | Jcode translation error | 4-byte integer containing translator error code, followed by a "tool-binding list." |
| 10 | VP translation error | 4-byte integer containing translator error code, followed by a "tool-binding list." |
| 11 | Invalid executable format | Name of tool containing invalid executable, followed by a "tool-binding list." |
| 12 | Could not resolve constant fixup | Constant fixup string which could not be resolved, followed by a "tool-binding list." |

# 3. Process Management

In intent, applications are comprised of processes. A process is made up of one or more threads, a thread being the route through the executable code in a series of tools.

In other systems threads are commonly used for the following reasons:

- It is usually faster to create or destroy threads than processes.
- Threads can be used to share certain types of data, such as open files, signal tables, etc.
- Threads from the same application often operate within a single memory space, allowing them to share data more efficiently. This occurs even on operating systems which generally compel memory protection between processes.

In intent, processes are able to usurp many of these advantages, so that there are not the same benefits in the use of threads instead of processes.

- The creation of processes in intent is very lightweight and memory efficient.
- In intent, memory protection between processes is not enforced. Thus processes running upon the same processor can access one another's data.
- Like other operating systems, intent supports threads which allow the sharing of system resources such as the file table, signal table, etc. However, in intent these threads are effectively normal processes with some additional attributes.

The kernel contains numerous process management functions, offering the capacity to create and destroy processes, to switch them between different 'states,' to change their scheduling characteristics and to acquire information about them.

## 3.1 Process States

Processes in real time intent can exist in one of several states, defined as follows:

- **NON-EXISTENT**
  This can be regarded as a pseudo-state, since a real intent process never actually exists in this state. A 'non-existent' process has not yet been created, or has been deleted. This 'state' is defined only so that the complete life-cycle of a process can be described.
- **DORMANT**
  A process in this state is either not yet ready to execute, or has already completed.
- **SUSPEND**
  A process in this state has been halted in its execution by another process.
- **SLEEP**
  A process in this state cannot execute because it has voluntarily gone to sleep, and is waiting to be woken up after a set amount of time or by another process.
- **SLEEP & SUSPEND**
  A process in this state has had a suspend operation performed upon it while already in the SLEEP state. It is therefore waiting both to be resumed and to be woken.
- **RUN**
  A process in this state is currently running. Only one process may exist in this state at any particular time.
- **READY**
  A process in this state is neither suspended or sleeping, and is therefore ready to execute, but is prevented from doing so because a higher or equal priority process is running.

The interrelation of the different process states is shown in Diagram 2.

Functions that directly modify the state of a process may be divided into two groups. One group operates upon the calling process, while the other group is called from one process to operate upon another. Only a process in the RUN state is capable of performing an operation, and only one process may be in this state at one time. Thus, a function to change the state of the running process can only be called by a process to modify its own state.

Changes to the state of a process may be effected by the primitive operations described below. The state of the calling process, among others, may be indirectly modified by functions such as the synchronisation functions (see later section on “Process Synchronisation”), but only through use of these same primitive operations. Diagram 2 shows the functions which are available for management of processes, and for control of their scheduling characteristics.

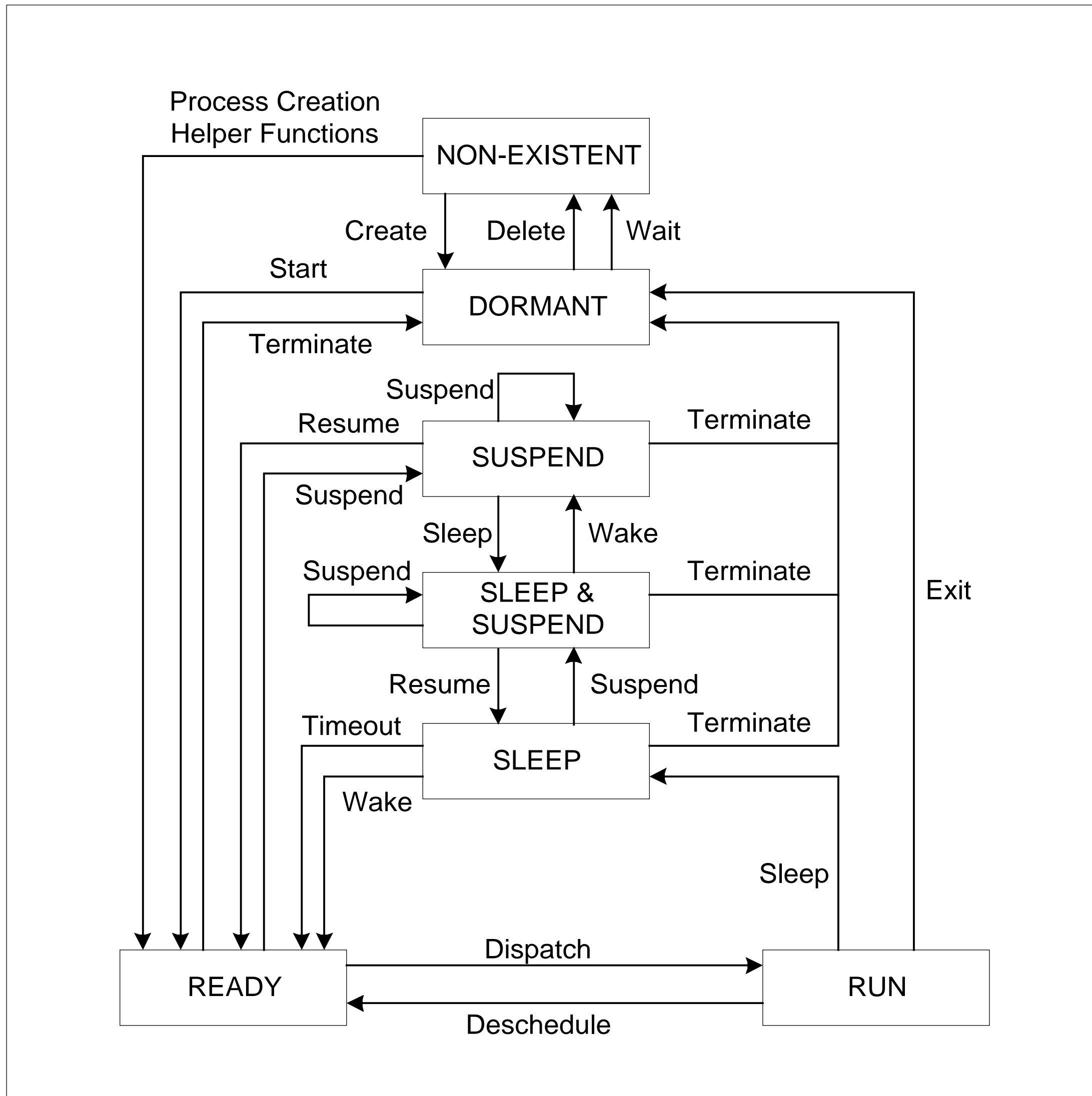


**Diagram 2: Process State Diagram**

The interrelation of the process states is described in even greater detail in Diagram 3 below.

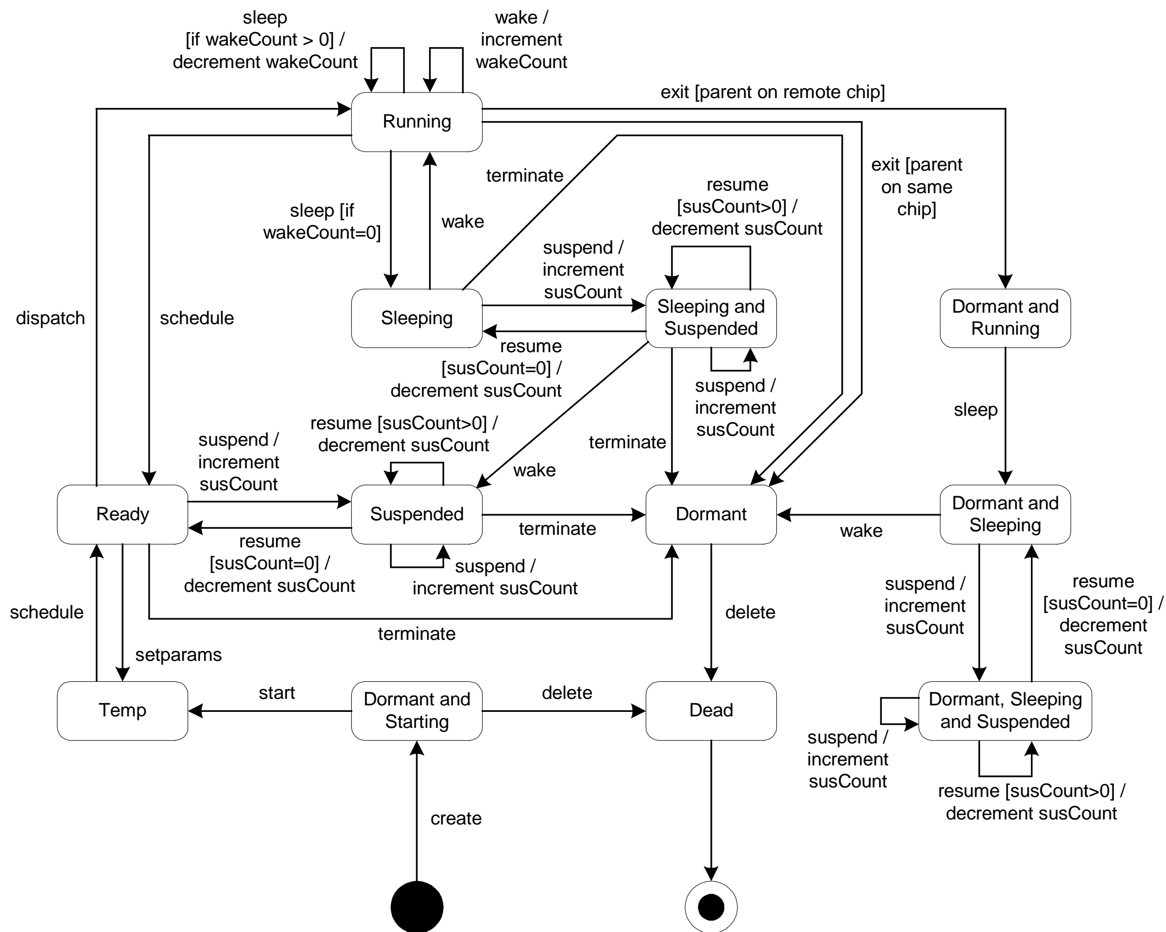


**Diagram 3 : Process State Diagram – Detailed**

# 3.2 Process Management Function Descriptions

## 3.2.1 Create a process - sys/kn/proc/create

This function allocates and initialises the process state, including the stack, global data area, file descriptors, environment list, etc. The main tool of the process is opened at this point. When this function returns, therefore, the main tool will be fully bound and ready to be executed (excepting any virtual or virtual+fixup *qcall*s it contains, which are bound at run-time).

This function moves the process from the NON-EXISTENT state to the DORMANT state. The process will not, however, begin running until it has been started (using *sys/kn/proc/start).* The function takes as parameters a pointer to a spawn structure, which is used to specify a process. The structure includes the name of the main tool, any command line parameters, any environment variables, and any file descriptors which are to be initially open.

The *sys/kn/proc/create* function also takes a pointer to a process control block (pcb) structure which contains the scheduling parameters for the process. If NULL is specified then the pcb structure of the calling process will be used.

More detailed descriptions of the spawn structure and the pcb structure may be found in "Data Structure Definitions" section later in this document. An account of the kernel functions for creating and modifying spawn structures may be found later in this section.

Where the function has completed successfully, the process identifier of the new process will be returned in the form of a positive integer. In the case of a failure, the function will return a negative integer. It should be noted that the values returned by this function do not correspond to the normal error code values.

The errors returned by this function are as follows:

| | |
|---|---|
| -1 | Invalid data memory object specified |
| -2 | Invalid stack memory object specified |
| -3 | No main tool specified |
| -4 | Failed to open/bind main tool |
| -5 | Tool specified does not have f_main bit set |
| -6 | Could not allocate global data area |
| -7 | Could not allocate stack |
| -8 | Could not allocate PID |
| -9 | Could not allocate scheduler data structure |
| -10 | General ‘out of memory’ error |
| -11 | Could not retrieve parent’s PCB block |

The process identifier is described in more detail later in this section.

### 3.2.2 Delete a process - sys/kn/proc/delete

This function deletes the specified process, and automatically frees not only the resources which were previously allocated by *sys/kn/proc/create*, but also some of the resources allocated by the process during its execution.

These include the stack, global data area, main tool reference, file descriptors and environment variables for the specified process. The function operates on a process in the DORMANT state, and therefore cannot be called by a process upon itself.

If any child processes belonging to the process being deleted are in the DORMANT state, they are also deleted. If any child processes belonging to it exist but are not in the DORMANT state, they become children of process ID 0, the *init* process, which should delete them in the correct manner when they terminate.

This function returns 0 if successful, or an error code indicating the error if it failed.

### 3.2.3 Start a process - sys/kn/proc/start

This function starts the specified process. It operates on a process in the DORMANT state, and moves it to the READY state. The priority of the process is set to the value specified in the initial pcb given when the process was created.

*sys/kn/proc/start* calls are not queued. In other words, if *sys/kn/proc/start* is called on a process which is not in the DORMANT state, the request is ignored, and an error code is returned. If the specified process later becomes DORMANT, it will not be immediately restarted due to this function call.

### 3.2.4 Stop the execution of a running process - sys/kn/proc/exit

This function is called by a process to end its own activity. It moves the calling process from the RUN state to the DORMANT state. From this DORMANT state it may be restarted or deleted by another process. The *sys/kn/proc/exit* function takes an integer as its parameter. This integer is returned to the process which calls *sys/kn/proc/delete* on this process. This function never returns to the caller, which, in the case of a successful completion, is rendered dormant.

Information about the process is reset whenever the process is moved to the DORMANT state. This information includes the contents of the pcb structure and the program counter, which gives the memory address for the instruction currently undergoing execution. These are reset to the values that were initially specified when the process was created. As a consequence, if the process has changed its scheduling parameters during execution by using the *sys/kn/proc/setparams* function, the new values are not retained when the process exits. If the process is restarted at a future point, its values return to their initial settings.

### 3.2.5 Terminate a process - sys/kn/proc/terminate

This function terminates the specified process, moving it from its current state into the DORMANT state. The function may not operate upon a process in the RUN state, or a process which is already in the DORMANT state. If the caller specifies a process in either of these states, an error is returned and no action is taken. This function is often used to induce abnormal termination in the case of a system malfunction.

An 'interrupt-safe' version of this tool, *sys/kn/int/proc/terminate*, is also available. *sys/kn/int/proc/terminate* can safely be called from inside an interrupt. Further details are available in the later section on “Interrupt Handling.”

### 3.2.6 Sleep for a specified time - sys/kn/proc/sleep

This function causes the calling process to block until it is 'woken' by another process or an interrupt handler, until the specified timeout expires, or until an unblocked signal becomes pending. Another process may wake the sleeping process by using the *sys/kn/proc/wake* function. An interrupt handler may wake it by using the *sys/kn/int/proc/wake* function.

Any queued calls to *sys/kn/proc/wake* will cause this function to return immediately.

The timeout parameter is a 64-bit quantity specified in nanoseconds, and should be -1 if no timeout is desired. The timeout is the period during which the process will wait to be woken before abandoning its SLEEP state.

If *sys/kn/proc/wake* or *sys/kn/int/proc/wake* is called before the period of time specified by the timeout parameter elapses, *sys/kn/proc/sleep* will complete normally, and a value of 0 will be returned. If a signal becomes pending for the process while it is asleep, *sys/kn/proc/sleep* returns EINTR. If a callback is pending when *sys/kn/proc/sleep* is called, or one becomes pending while the process is asleep, EINTR is returned. If none of the above conditions become true before the timeout period elapses, ETIMEDOUT is returned.

It is possible for several of these conditions to occur simultaneously. For example, it is possible for the timeout to expire just as the process is being woken. Alternatively, a signal might be sent to the process at the same time as the expiry of a timeout, or the waking of a process.

If a combination of these conditions occurs, the call to *sys/kn/proc/wake* is always considered to have occurred before the arrival of a signal, a callback or a timeout. Similarly, a timeout is always considered to have occurred before a signal or a callback. Since there is no way to discriminate between signals and callbacks, they have equal precedence. Thus, EINTR will be returned if a signal

or callback occurred, but there was no timeout or call to *sys/kn/proc/wake.* ETIMEDOUT will be returned if a timeout occurred but no call was made to *sys/kn/proc/wake.* If a call was made to *sys/kn/proc/wake* or *sys/kn/int/proc/wake,* zero will be returned, regardless of the occurrence of the other conditions.

Since the *sys/kn/proc/sleep* function may only be called by a process upon itself, calls to *sys/kn/proc/sleep* cannot be nested. Once in the SLEEP state, the calling process cannot place subsequent calls to the function, and thus create 'layers' of sleep, each needing to be dispelled by a different call to the *sys/kn/proc/wake* function. It is possible, however, for another process to call *sys/kn/proc/suspend* specifying a task which is in the SLEEP state. This will result in the specified process changing to the SLEEP&SUSPEND state.

One can determine whether a callback was taken during a sleep by examining the value returned by *sys/kn/callback/occurred*. The value of this flag is cleared on entry to *sys/kn/proc/sleep* if callbacks are enabled, and is set if callbacks are processed. It should be noted that if callbacks are completely disabled, the value of this flag is not affected by a call to *sys/kn/proc/sleep*.

If the callback flag state (see the description of *sys/kn/callback/setflag* in the section on “Callbacks”) is 2, then a callback being posted to the sleeping process will cause it to wake. The callback, however, will not be taken immediately. Instead it will become pending. This condition can be tested by calling *sys/kn/callback/pending*.

### 3.2.7 Wake a sleeping process - sys/kn/proc/wake

This function releases the specified process from its SLEEP state. A process may not specify itself in this function call.

If the specified process is not in the SLEEP state, the *sys/kn/proc/wake* request will be queued, so that the next time the specified process calls *sys/kn/proc/sleep,* the call will return immediately and the process will be 'woken.'

This function can be called when interrupts are off, and will return with interrupts in the same state as when it was called. However, the operation of calling *sys/kn/proc/wake* cannot be assumed to be atomic, as the process being woken may run, with interrupts on, before this function returns.

An 'interrupt-safe' version of this tool, *sys/kn/int/proc/wake*, is also available. *sys/kn/int/proc/wake* can safely be called from inside an interrupt. Further details are available in the later section on “Interrupt Handling.”

### 3.2.8 Suspend the operation of a process - sys/kn/proc/suspend

This function is called by one process to operate upon another, moving it into the SUSPEND state.

If the specified process is in the SLEEP state, this function moves it into the SLEEP&SUSPEND state. If, while in this state, the process is the subject of a *sys/kn/proc/wake* function call, it will be moved into the SUSPEND state. If the process is the subject of a *sys/kn/proc/resume* function call while in the SLEEP&SUSPEND state, it will be moved back to the SLEEP state.

A process may not specify itself as the parameter to the *sys/kn/proc/suspend* function.

*sys/kn/proc/suspend* calls are queued. If a process is the subject of a series of *sys/kn/proc/suspend* function calls, it requires an equal number of *sys/kn/proc/resume* function calls to release it from the SUSPEND state. A suspend count is maintained for each process, which is incremented by each successful call to *sys/kn/proc/suspend* and decremented by each successful call to *sys/kn/proc/resume*.

The fact that a process is in the SUSPEND state does not affect how it is allocated resources such as semaphores and mutexes.

An 'interrupt-safe' version of this tool, *sys/kn/int/proc/suspend*, is also available. *sys/kn/int/proc/suspend* can safely be called from inside an interrupt. Further details are available in the later section on “Interrupt Handling.”

### 3.2.9 Decrement the suspend count for a process - sys/kn/proc/resume

This function decrements the suspend count described in the section on the *sys/kn/proc/suspend* function. If the suspend count reaches zero, the process has its SUSPEND state removed. This does not necessarily mean that the process becomes READY, since it may just move the process from the SLEEP&SUSPEND state to the SLEEP state.

A process may not specify itself as the parameter to this function.

This function does not allow the suspend count to fall below zero. If this function is called on a function which is not in the SUSPEND state and thus has a suspend count of zero, an error is returned.

This function can be called when interrupts are off, and will return with interrupts in the same state as when it was called. However, the operation of calling *sys/kn/proc/wake* cannot be assumed to be atomic, as the process being woken may run, with interrupts on, before this function returns.

An 'interrupt-safe' version of this tool, *sys/kn/int/proc/resume*, is also available. *sys/kn/int/proc/resume* can safely be called from inside an interrupt. Further details are available in the later section on “Interrupt Handling.”

### 3.2.10 Yield CPU - sys/kn/proc/deschedule

This function moves the calling process to the end of the scheduler list for the appropriate priority level, and invokes the scheduler to dispatch another process. This is discussed in greater detail in the section on process scheduling, later in this document.

The calling process is moved from the RUN state to the READY state.

### 3.2.11 Cause scheduling to take place - sys/kn/proc/ideschedule

This function is solely for use by PIIs as described in PII development documentation. Use by any other code is likely to cause unpredictable behaviour.

This function is called by the PII when an interrupt handler has indicated that it requires a scheduler operation. It should be called in a normal CPU context.

*sys/kn/proc/ideschedule* causes timers to be processed. If there are any processes which have been woken or rescheduled in some other way during the interrupt service, and which have a higher priority than the process which was running at the time of the interrupt, then the function allows them to run immediately.

The function should be called within the context of the process which was running at the time of the interrupt. The calling process is moved from the RUN state to the READY state. It will be run later when it becomes the highest priority runnable process in the system. It is possible that this may occur immediately.

### 3.2.12 Alter the values in the process control block - sys/kn/proc/setparams

This function changes the scheduling parameters of the specified process. A process may call *sys/kn/proc/setparams* to operate upon another process, or upon itself.

The first parameter to *sys/kn/proc/setparams* is a pointer to the identifier of the process that must be modified. The second parameter is a pointer to a process control block structure which contains the parameters to be set.

If the process is not running, and its priority is modified so that it becomes the highest priority runnable process, a context switch will be made so that this process can run. If the process is running, and its priority is modified so that it is no longer the highest priority runnable process, a switch will be made to the new highest priority runnable process.

An 'interrupt-safe' version of this tool, *sys/kn/int/proc/setparams*, is also available. *sys/kn/int/proc/getparams* can safely be called from inside an interrupt. Further details are available in the later section on "Interrupt Handling."

### 3.2.13 Acquire information about a process - sys/kn/proc/getparams

This function returns information about the specified process. A process may call *sys/kn/proc/getparams* to acquire information about another process, or about itself. The second parameter to *sys/kn/proc/getparams* is a pointer to a process control block structure. The *sys/kn/proc/getparams* function will fill this structure with the relevant parameters.

### 3.2.14 Wait for a child process to stop or terminate, and delete it if it has terminated - sys/kn/proc/wait

This function checks whether the specified child process has terminated or stopped. If no process was specified, the function checks all child processes of the caller and operates upon the first of these that has stopped or terminated.

If the specified process has terminated, its exit status is stored in the location specified by the input pointer, the process is deleted, and its process ID is returned.

If the specified process has stopped due to a job-control signal, information about the process's current state is stored in the specified location. Its process ID is then returned. The information stored may be examined using the WIF* macros. These macros can only be used to return information about stopped processes if the WUNTRACED flag is specified on input.

If the input integer specifies a process that has neither terminated nor stopped, then the action of the function will depend upon the flag parameter. If the WNOHANG flag is specified, this ensures that the calling process does not 'hang.' Instead of blocking, the process returns ECHILD, to indicate that it has not found a child process of the appropriate type. If the WNOHANG flag is not specified, the calling process will try to block until the specified child process terminates or stops due to a job-control signal.

Notification of the death of a child process is performed using the SIGCHLD signal. Thus, if the calling process has its signal mask adjusted so as to block this signal, it cannot wait for the child process to terminate, since this would cause it to block indefinitely.

Similarly, if the signal handler for the SIGCHLD signal is not set to SIG_DFL, the calling process cannot block waiting for the child process. If the handler is set to SIG_DFL, this ensures that the it will perform the process's default action in response to the signal. In the case of SIGCHLD, the default action is to ignore the signal.

In both of these cases, ECHILD is returned.

It should be noted that this function will not reap child processes which have been created but not yet started. This removes the danger of a newly-created process being accidentally deleted due to the occurrence of a SIGCHLD after the process has been created and before it has been started.

More details on SIGCHLD, signal handlers and signal masks may be found in the "Signals" section later in this document.

### 3.2.15 Wait for a child process to terminate - sys/kn/proc/chld

This function behaves in the same way as *sys/kn/proc/wait*, except that it does not delete the child process. Instead, it is the caller's responsibility to call *sys/kn/proc/delete*, and thus ensure that the child process is properly deleted.

### 3.2.16 Change the priority of the calling process - sys/kn/proc/chpri

This function changes the priority of the calling process to the specified value. The previous priority value is returned. If the 'new priority value' is given as -1, the priority is not changed.

If due to the changes caused by this function the calling process ceases to be the highest priority runnable process in the system, it is pre-empted by the new highest priority process.

### 3.2.17 Return the priority of the calling process - sys/kn/proc/getpri

This function returns the priority of the calling process.

### 3.2.18 Set the parent process of the specified process - sys/kn/proc/chppid

This function changes the parent process ID of the specified process to the given value. The process IDs given for the process and for its 'new parent' must both be valid and existent. After this function has operated upon a process, the system considers the process to be the child of the specified 'new parent.' The ID of the process's 'previous parent' is returned.

This functionality allows the programmer to choose which process will receive SIGCHLD notification when a process dies. The 'new parent' should be prepared to receive SIGCHLD signals when its new 'child' terminates, and to reap it at this point in the appropriate manner. If process ID 0 is set as the 'new parent,' it will automatically reap any terminating child processes for which it receives a SIGCHLD signal, regardless of whether it started them.

### 3.2.19 Sleep until event occurs, storing any spurious wakes - *sys/kn/proc/devsleep*

This function is for use primarily by device drivers. It is used to wait until the occurrence of an event specified by the caller, or until an abnormal termination condition arises.

*sys/kn/proc/devsleep* is called with interrupts off, and returns with interrupts off. This function assumes that the caller has set up a callback function and signals to be enabled or disabled, according to the caller's preference.

*sys/kn/proc/devsleep* returns when the specified timeout occurs, when an unblocked signal becomes pending, or when the user-specified event occurs. The occurrence of the event is determined by calling the callback function specified in the parameter list. The callback function takes a single pointer parameter (specified by the caller to this function), and returns an integer. If this integer has a non-

zero value, the event has occurred. The callback function is called with interrupts off, and is expected to operate entirely without re-enabling interrupts.

If the calling process is the subject of any calls to *sys/kn/proc/wake* during this period, the number of calls is stored, and restored when this function finally returns, such that the wake count of the process when this function returns will be the sum of the count on entry to this function and the number of calls to *sys/kn/proc/wake*, minus 1. (The count is reduced by one for the call to *sys/kn/proc/wake* which caused this function to return, and which may have been due to a signal).

### 3.2.20 Add routines to atexit list - sys/kn/proc/atexit

This function is used to add a routine to the 'atexit list.' All routines on this list are executed when the process from within which *sys/kn/proc/atexit* was called exits.

Passing -1 in place of user data causes all nodes referencing the routine specified by the first input pointer parameter to be removed from the atexit list. The function returns success if this routine is not present on the atexit list.

*sys/kn/proc/atexit* may be called from within a routine on the atexit list.

## 3.3 Process Creation Helper Functions

Besides these general process management functions, intent offers several 'process creation helper functions' which facilitate process creation by combining the functionality of *sys/kn/proc/create* and *sys/kn/proc/start.*

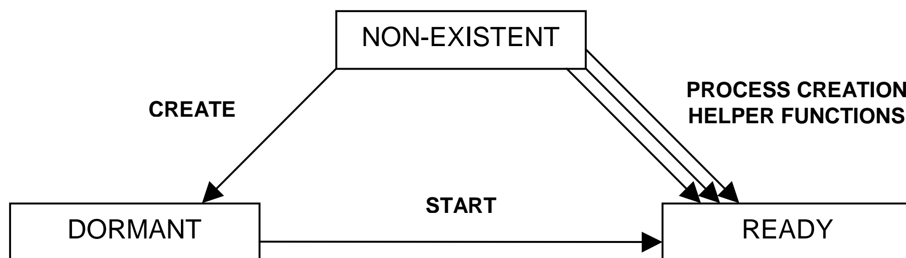


**Diagram 4 - Process Creation Helper Functions**

In addition, different helper functions allow the creation of processes on remote processors, or abet the even distribution of processes across the intent network.

### 3.3.1 Create and start a process on the same processor as the calling process - sys/kn/proc/exec/local

This function creates and starts a process on the same processor as the calling process, using the specified parameters.

### 3.3.2 Create and start a process on a specified processor - sys/kn/proc/exec/remote

This function creates and starts a process on the specified processor, using the specified parameters.

It is possible that the calling process may be a local process, that is to say, one with a process ID in the non network-unique range. If this is the case, or if the spawn structure passed in has a SPAWN_LOCAL field, 0 will be returned as the child's process ID. In such circumstances, the parent

process will not be able to communicate with the child, even to wait for the child process or send it signals.

### 3.3.3 Create and start a process on a processor chosen by the kernel - sys/kn/proc/exec/any

This function creates and starts a process using the specified parameters. The intent kernel is permitted to select the processor upon which the process is started. This selection is made using the kernel's load balancing routines.

intent makes use of a 'load balancing' algorithm in order to ensure that within a network some processors do not lie idle while others are overburdened. Taking into consideration the speed of execution for each processor, and the number of processes running upon each, intent is able to assign new tasks to the processors that are less 'busy' and thus capable of executing the new processes more swiftly.

## 3.4 Spawn Structure Functions

The spawn structure is a data structure integral to the 'spawning' of new processes. A spawn structure holds all the data necessary for the creation of a new process, apart from scheduling parameters. In order to create a new process, a pointer to the relevant spawn structure should be passed as a parameter to *sys/kn/proc/create*. A more detailed account of spawn structures may be found later in this document, in the section on "Data Structure Definitions."

The functions for the creation of spawn structures may be found below. The structures thus created are not 'consumed' or modified by calls to *sys/kn/proc/create*, may be used more than once, and must be freed when no longer required.

### 3.4.1 Create a spawn structure - sys/kn/proc/spawn/make

This function creates a spawn structure from the supplied parameters. The *sys/kn/proc/spawn/make* function is passed all data necessary for the creation of the prospective child process, except for scheduling parameters, and unites this information within a spawn structure. This structure contains the most commonly used spawn structure record types, and may then be used as a parameter to sys/*kn/proc/create*.

This function takes a number of pointer parameters, which specify:

- The name of the main tool to be run for the child process. This tool must have the TF_MAIN flag set in its tool header to indicate that it is a 'main tool.'

- A pointer to an array of other pointers, which indicate the strings that will be supplied to the new process as its argv array. If this parameter is NULL, the child process receives no *argv* array. The array of pointers must be terminated by a NULL pointer, to mark the end. The input array is copied by this function rather than the original being shared.

- A pointer to the name of the memory object to use for the stack of the child process. If NULL is specified, the default stack memory object is used.

- A pointer to the name of the memory object to use for the data of the child process. If NULL is specified, the default data memory object is used.

- A pointer to a block of data to be copied into the global data area of the child process when it is created.

Further integer parameters specify:

- The size of the area to be copied into the global data area.
- A flag that specifies whether the prospective child process should inherit file descriptors from the calling process. If this flag has the value 0, file descriptors are not inherited. If the flag is 1, all file descriptors without the FD_CLOEXEC flag set are copied into the spawn structure. Before a spawn structure is used to create a process, these file descriptors can be modified or deleted by using the *sys/kn/proc/spawn/modf* function. The function may also be used to add new file descriptors to the spawn structure.
- A flag that specifies whether the child process should inherit environment variables from the calling process. If this flag is 0, environment variables are not inherited by the child process. If the flag is 1, all environment variables in the calling process are copied to the structure, and inherited by the child.

Much of the information contained within the spawn structure can be modified, or new data added, before process creation, using the functions detailed below.

### 3.4.2 Create a spawn structure (exec-style) - sys/kn/proc/spawn/emake

This function creates a spawn structure from the supplied parameters, which may then be used as a parameter to *sys/kn/proc/create*. The *sys/kn/proc/spawn/emake* function differs from the *sys/kn/proc/spawn/make* function in that it uses an interface similar to that of *exec*, the POSIX function for creating processes.

The parameters taken by the function are as follows:

- A pointer to a string specifying the name of the main tool to be run for the child process. This tool must have the TF_MAIN flag set in its tool header.
- A pointer to an array of other pointers, which point to the strings that will be supplied to the new process as its *argv* array. If this parameter is NULL, the child process receives no *argv* array. The input array is copied by this function rather than the original being shared.
- A pointer to an intent environment object containing environment nodes to be copied into the child's environment. A description of environment nodes may be found in the section on "Data Structure Definitions."
- A flag specifying whether to inherit file descriptors from the calling process. If this flag has the value 0, file descriptors are not inherited. If the flag is 1, all file descriptors without the FD_CLOEXEC flag set are copied into the spawn structure. They may be modified or deleted before the spawn structure is used to create a process using the *sys/kn/proc/spawn/modfd* function. File descriptors may also be added to the spawn structure using this function.

The structure returned should be freed by the calling process when it is no longer required, using *sys/kn/mem/free*.

### 3.4.3 Modify file descriptor in spawn structure - sys/kn/proc/spawn/modfd

This function modifies a spawn structure. Besides the pointer to the relevant spawn structure, this function takes two integer parameters. The second integer parameter (fd1) is the file number of the file descriptor to be changed in the child process. The first integer parameter (fd0) is the file number of a file descriptor existing in the parent process.

The spawn structure is searched for a file descriptor record with file number fd1. If found, this file descriptor record is removed. If the file descriptor with file number fd0 is open in the calling process, a record is added to the spawn structure duplicating that file, but giving it the file number fd1. Thus, the function permits a specified file descriptor, copied from the calling process, to replace a descriptor contained within the spawn structure.

The spawn structure returned by the function may be different from the input structure, in which case the input and output spawn structure pointers will differ, and the old structure will still exist unmodified. Both structures need to be freed using *sys/kn/mem/free* when they are no longer needed.

### 3.4.4 Add global data initialisation in spawn structure - sys/kn/proc/spawn/modglobs

This function modifies a spawn structure, inserting global data initialisation record. If there is already a global data initialisation record in the spawn structure which is large enough for the specified data, a pointer to it is returned to the caller.

If there is not, a new spawn structure is created which is identical to the input spawn structure except that it contains a global data initialisation record large enough for the specified amount of data.

The spawn structure returned by the function may be different from the input structure, in which case the input and output spawn structure pointers will differ, and the old structure will still exist unmodified. Both structures need to be freed using *sys/kn/mem/free* when they are no longer needed.

It should be noted that if one is using C, the function *sys/kn/proc/spawn/modglobs* returns a pointer to the modified spawn structure. The pointer to the global data initialisation area will be returned in the location indicated by `result`. In addition when using C, the function *sys/kn/proc/spawn/modglobs* returns two values, so it is necessary to use the multiple return registers facility of the Elate C compiler.

### 3.4.5 Add SPAWN_PARENT record to spawn structure - sys/kn/proc/spawn/modparent

This function modifies a spawn structure, inserting a SPAWN_PARENT record if one does not already exist. If there is already a SPAWN_PARENT record, a pointer to the original spawn structure is returned to the caller.

If there is not, a new spawn structure is created which is identical to the input spawn structure except that it contains a SPAWN_PARENT record.

The spawn structure returned by the function may be different from the input structure, in which case the input and output spawn structure pointers will differ, and the old structure will still exist unmodified. Both structures need to be freed using *sys/kn/mem/free* when they are no longer needed.

### 3.4.6 Add SPAWN_STACK record to spawn structure - sys/kn/proc/spawn/modstack

This function modifies a spawn structure, inserting a SPAWN_STACK record if one does not already exist. If the structure already contains a SPAWN_STACK record, the size of stack requested is updated if necessary, and a pointer to the original spawn structure is returned to the caller.

If a SPAWN_STACK record does not exist, a new spawn structure is created. This is identical to the input spawn structure except that it contains a SPAWN_STACK record.

The spawn structure returned by the function may be different from the input structure, in which case the input and output spawn structure pointers will differ, and the old structure will still exist unmodified. Both structures need to be freed using *sys/kn/mem/free* when they are no longer needed.

### 3.4.7 Add SPAWN_SIGMASK record to spawn structure - sys/kn/proc/spawn/modsig

This function modifies a spawn structure, inserting a SPAWN_SIGMASK record if one does not already exist.

If there is already a SPAWN_SIGMASK record, the signal mask is updated if necessary and a pointer to the original spawn structure is returned to the caller. If there is not, a new spawn structure is created. This is identical to the input spawn structure except that it contains a SPAWN_SIGMASK record.

The spawn structure returned by the function may be different from the input structure, in which case the input and output spawn structure pointers will differ, and the old structure will still exist unmodified. Both structures need to be freed using *sys/kn/mem/free* when they are no longer needed.

### 3.4.8 Add SPAWN_STACKLIMIT record to spawn structure - sys/kn/proc/spawn/modstklimit

This function modifies a spawn structure, inserting a SPAWN_STACKLIMIT record if one does not already exist.

If there is already a SPAWN_STACKLIMIT record, the size limit is updated if necessary and a pointer to the original spawn structure is returned to the caller. If there is not, a new spawn structure is created. This is identical to the input spawn structure except that it contains a SPAWN_STACKLIMIT record.

The spawn structure returned by the function may be different from the input structure, in which case the input and output spawn structure pointers will differ, and the old structure will still exist unmodified. Both structures need to be freed using *sys/kn/mem/free* when they are no longer needed.

### 3.4.9 Adds requirement to use local PID to spawn structure - sys/kn/proc/spawn/modlocal

This function modifies a spawn structure, inserting a record which requires the spawned process to use a local process ID. If there is already such a record in the specified spawn structure it is returned unchanged. If such a record does not already exist in the input spawn structure, a new spawn structure is created which is identical to the input spawn structure except that it contains a record to require use of a local process ID.

On a multiprocessor system, use of a local process ID is appropriate for processes which do not interact in any way with processes on other CPUs. Use of a local PID is compulsory for any process which is created during system boot before the multiprocessing server is running. This may be the case for certain device drivers.

The spawn structure returned by the function may be different from the input structure, in which case the input and output spawn structure pointers will differ, and the old structure will still exist unmodified. Both structures need to be freed using *sys/kn/mem/free* when they are no longer needed.

## 3.5 Spawn structure macros

In addition to the functions described above for dynamically creating spawn structures, Elate provides a set of macros which can be used to statically create template data for spawn structures in the body of tools at assembly time. These templates can then be copied out into RAM at runtime, onto the stack or into an allocated memory block. They can then be passed to any function which requires a spawn structure as a parameter, including *sys/kn/proc/create* and the spawn structure functions described earlier in this section.

- **spawnstart** defines the beginning of the spawn structure, and also its name. Each spawn structure template defined using the spawn structure macros has a different name, so that they can be referred to individually. This name can be used to refer to the structure template, and also to the offsets of the fields within the structure. This macro should only be used in the data section of the tool.

- **spawnend** marks the end of the template spawn structure. It must come after the corresponding *spawnstart* macro, with only *spawn* macros used in between.

- **spawn** should only be used between the *spawnstart* and *spawnend* macros. Each use of the **spawn** macro defines one record into the spawn structure template. The record type emitted is specified by the first parameter to the **spawn** macro. It is possible to use the symbolic names for the record types used in the spawn structure, with the SPAWN_ text removed from the start of their names. An full list of these record types can be found in the description of spawn structures in the “Data Structure Definitions” section later in this document.

The *spawn*/*spawnstart*/*spawnend* macros serve several purposes:

In the first place, they create a template spawn structure. At assembly time it is often impossible to know all of the data which needs to be put into the structure before it is used. Therefore, these macros create a template structure which contains most but not quite all of the necessary data at the correct locations.

The example below creates a "Stack object" record. This record needs to contain the address of the stack memory object to be used for the process being created. This address cannot be known at assembly-time. This part of the data is not filled in, therefore, but the record for the stack object is generated so that the information can be inserted later.

```
data
spawnstart tspawn
spawn NAME, 'sys/kn/cpu/server' ;Create a "toolname" record
spawn STACKOBJ ;Create a stack object record
spawn DATAOBJ ;Create a data object record
spawnend
```

The spawn structure macros also define a label with the name specified on the *spawnstart* macro line. This label points to the start of the template structure. The macros also define the size of the structure using the same name, with *_size* appended. In the above example, therefore, the label is *tspawn* and the size is *tspawn_size*.

Finally, the macros define assembler constants, which define the offsets from the start of the structure to particular items of data within the structure. In the above example, the offset of the stack object data is defined as *tspawn_stackobj*, and the offset of the data object data is defined as *tspawn_dataobj*. The main toolname record is entirely defined at assembly time, so it is not necessary for a constant to be defined for it.

The spawn structure is copied onto the stack, its undefined portions are filled in, and it is then ready for use. The structure template must be copied out of the tool body into a RAM area before any of its data fields are modified, since the tool body may be in ROM. However, the template structure may be complete, in which case it is not necessary to copy it to RAM. (If the structure consists of just a main tool record, for example, the structure will have no incomplete records.)

# 4. Process Management Data Structures

## 4.1 Process ID

In order to allow easy reference to the various processes within a system, int**e**nt has adopted the widely used concept of a "Process ID." int**e**nt process IDs are defined to be unsigned 32-bit integers. These cannot be confused with error codes, since the defined error code values are not valid process IDs.

The Process ID can be used to find an index to an entry in a table known as the "Process Table". Any functions which operate upon a process will use this ID as a parameter, to direct them to the correct address. The process table provides a mapping between each process ID and the scheduler data structure (SCHED_) for the related process. Access to the process table takes place via a well-defined, portable API. This level of abstraction ensures that applications will remain compatible with all future improvements to the kernel, and permits implementations of differing functionality.

The process table can be implemented with a fixed size, which is defined when the kernel is compiled. If at design time a reasonable upper bound is known for the number of processes which will need to be able to run upon the system at any one time, such an implementation offers more efficiency in terms of speed and memory than a more flexible strategy. The 'simple' process table is an example of such an implementation.

Thus, the simple process table might be suitable for an embedded system for which the maximum number of concurrent processes was known at design time. However, it supports only local processes and so is unsuitable for a multi-processor system which requires network-unique Process IDs (see below).

The fixed size of the table imposes a limit upon the number of processes which may run concurrently on the system. In most systems the table can be defined to be quite large, so that this limitation does not pose a problem. However, in the case of an embedded system, or any other system requiring a small footprint, the memory cost of a large table is undesirable. On many UNIX systems, the table is defined to have 32768 entries. If each entry is assumed to take up 4 bytes, the table may be expected to use 131072 bytes of memory (128kbytes).

Where it is not possible to predict how many processes will be running in a system at the same time, a more flexible solution is needed. In the case of desktop systems, set-top boxes where code is downloaded across a network, or systems where an unknown number of processes may be started dynamically, the hierarchical process table structure will be used. This design allows for the dynamic extension of the table as required, up to the physical limitations of memory, and is the default implementation of the process table in int**e**nt.

## 4.2 Hierarchical Process Table

The data structures used in the hierarchical process table have considerable similarity to the structure of "Page Tables" on an x86 microprocessor.

As can be seen in Diagram 5, below, these tables exist in a hierarchical structure. The top level table is called the "Process Table Directory." This contains pointers to the second level tables, which are known as the "Process Tables." If a particular process table does not exist, the corresponding entry in the Process Table Directory is 0.

Each process table is 4096 bytes in size, and contains 1024 entries, each 4 bytes long. Each entry contains a pointer to the data structures for the corresponding process, or a NULL pointer if the corresponding process does not exist.

The process table directory, which will contain a pointer to the first process table, has an initial size of 4 bytes. If more than 1024 process IDs are required, a new process table is created in which to store the pointers to the processes. Since the process directory table will not be large enough to contain the pointer to this new table, a new directory table, 4 bytes bigger, is created and the old one freed. The number of process IDs can be dynamically extended in this way as many times as physical memory constraints allow.

This is a significant improvement over the conventional fixed size table method. For example, if it is thought that an application might require over 4000 concurrent processes, it would be necessary to create a 16kbyte fixed size table, which would exist for the duration of the application. The hierarchical process table will take up only slightly over 4kbytes at start-up time, and will only allocate more memory as and when the extra process IDs are necessary, This memory is freed again as soon as it is no longer needed. Use of this method leads to a significantly lower memory requirement for at least part, and potentially all, of the life of the application.

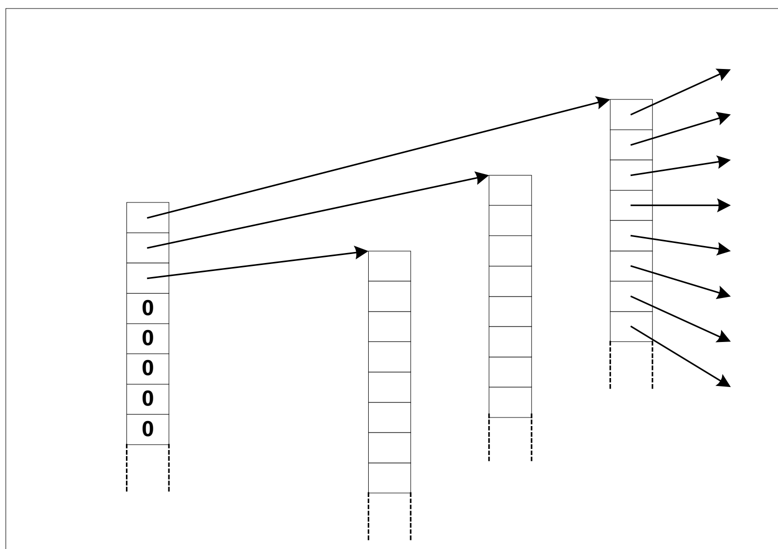


**Diagram 5: Process Table Structure**

The pointer to a specified process's data structures can be found in a quick and deterministic manner, by using the following algorithm:

1. Take the process ID and divide it by 1024, using integer division. This gives the index into the Process Table Directory.

2. Use this number to find the pointer to the correct process table. If the pointer is NULL, the specified process ID is invalid.

3. Using the remainder from the above division, find the correct entry in the process table. If the entry is NULL, the specified process ID is invalid. Otherwise, the entry contains a pointer to the data structures for the specified process.

By using powers of 2 for the sizes of the process tables, the division described above can be performed using SHIFT and AND operations, making it extremely fast to execute.

## 4.3 Network-Unique Process IDs

When an int**e**nt system spans a network of several processors, it is useful to be able to refer to a process on another machine. In order to be able to use the process ID for this purpose, it must be guaranteed that the ID will refer to no more than one process throughout the whole network.

Process identifiers are not network-unique by default, since not all processes require a network specific ID. Many processes, including most created by device drivers or the kernel, are not referred to by code outside the processor upon which they run. Such processes need only 'local' IDs. Application processes, on the other hand, may be distributed to processors other than those upon which their parent processes run, and for this reason generally need their IDs to be network-unique.

If the kernel on a specific processor needs to assign network-specific IDs, it will allocate a particular range of ID values which will then be reserved for this purpose. This allocation is made with the agreement of the kernels running upon the other processors in the network, which subsequently will not attempt to assign values from this range as identifiers for their own processes. If the first kernel exhausts the values in the allocated range, it may request an additional range.

Since communication with the kernels on the other processors of the network is a prerequisite for the allocation of a network-unique process ID range, network-unique IDs may not be allocated upon a processor that is not yet connected to the network. Care should be taken by the programmer and system integrator to ensure that processes created prior to their processor's connection to the network are assigned 'local,' rather than network-unique, IDs.

In addition, network-unique IDs are only supported by systems that employ int**e**nt's default implementation of the process table, the hierarchical table model. The simple implementation entailing the single, fixed size look-up table, supports only local process IDs.

The use of a local process ID entails several restrictions. A process with a local ID may create a child process on another processor. However, when this child terminates, it will be unable to notify its parent of this by means of a SIGCHLD signal, in the usual manner. (Details of SIGCHLD may be found in the section on "Signals" later in this document.) Instead, it is necessary to reset the 'parent process ID' of the child process to 0, so that they will be automatically deleted after termination. This can be achieved through the use of the function *sys/kn/proc/chppid* (see section on "Process Management Function Descriptions"). Under these circumstances the parent cannot use *sys/kn/proc/wait* for the child, since this would cause the parent to block indefinitely, waiting for the SIGCHLD message it was unable to receive.

## 4.4 Processor ID Handling Functions

All accesses to the process table should be made using the functions in the *sys/kn/proc/pid* directory. This provides scope for the system to provide multiple implementations. These functions can be used to look up a specific process ID in the process table, to allocate a new process ID, to allocate ranges of network-unique process ID values, to return the current processor number or the ID of the calling process, and so forth.

A system which uses only local process IDs may still implement the process ID handling functions in this directory. However, those functions related to the allocation of network-unique ID ranges would always need to fail. Any requests for allocation of network-unique process IDs should be mapped onto local process ID requests. An implementation of this type could not run on a processor that was part of a larger network.

Some of these functions are for internal use within the kernel, and thus are not listed in this document. *sys/kn/proc/pid/get* and *sys/kn/proc/pid/enumerate*, however, are described below.

### 4.4.1 Gets the PID of the calling process - sys/kn/proc/pid/get

This function takes no input parameters, and returns the processor ID of the calling process.

### 4.4.2 Return an array of all valid PIDs - sys/kn/proc/pid/enumerate

This function returns an array containing all the PIDs which correspond to processes on the caller's processor, effectively a list of all the processes in the local system. Some of these PIDs may, however, correspond to processes which have exited but have not yet been reaped, or which have not yet been started.

The array should be freed via a call to *sys/kn/mem/free* once it is no longer required.

Since this tool returns two values, if it is called it from C it is necessary to use the multiple return registers facility of the Elate C compiler.

# 5. Process Scheduling

## 5.1 The intent Scheduler

The scheduler is the most time critical part of the intent kernel, and is run whenever a task-switching operation is required. This may occur when a high priority process becomes runnable, or when the current timeslice expires, at a priority level where timeslicing is enabled. This may also occur during co-operative scheduling operations, such as when a process goes to sleep.

The intent scheduling model has two hundred and fifty-six priority levels, each of which can contain an unlimited number of processes simultaneously. Like schedulers in other operating systems, the intent scheduler is divided into two basic parts. The first is the dispatcher, a simple mechanism which runs the first process in a list of schedulable structures representing the processes awaiting execution. The second part governs scheduling policy, and determines the order of the processes within this list.

### 5.1.1 Contexts

As the scheduler switches between the processes demanding processor time, it is essential that processes should be able to resume execution in a condition unaltered during their period of suspended animation. For this reason it is necessary to save 'contexts,' records of the states of the program counter, stack pointer and so on, all of which define the state of a process at a particular moment in time. On a simple system, this usually consists only of the processor's native registers. On a more complex processor, this may include the state of the memory management unit, and the state of other parts of the CPU.

The code to save these contexts is processor specific, and parts of it are often written in native code. Functions devoted to initialising, storing and restoring contexts can be found in the CPU Isolation Interface (CII).

Descriptions of these functions may be found in the relevant developer documents in the build, notably *sys/cii/api.html*. Further description of CII functionality may also be found in *The System Programmer's Guide*.

## 5.2 The Dispatcher

The dispatcher is common to all scheduling models. Each schedulable structure included upon the run queue corresponds to an intent process. These structures each contain a small header, which is comprised of a list node, the process's priority level, a pointer to the process's global data area, the process's suspend counter, its wake counter, and some information about the signal state of the process. After the header, there is a data block which is maintained by the tool implementing the system's scheduling policy. The contents of this block are dependent upon the demands of the scheduling policy of the individual system, and therefore may not be described in detail in this document.

When the dispatcher is invoked, it takes the node from the front of the run queue, and dispatches the corresponding process. It also updates the pointers in the "scheduler pointer table" and the "scheduler bit table".

These two tables are used to provide deterministic searching and insertion into the run queue. The "shortcut pointer table" contains 256 pointers, one for each priority level. Each pointer points to the first node in the run queue of the corresponding priority. If there are no nodes in the run queue for a particular priority level, the pointer in the table entry for that priority contains zero.

The shortcut table is illustrated in Diagram 6.

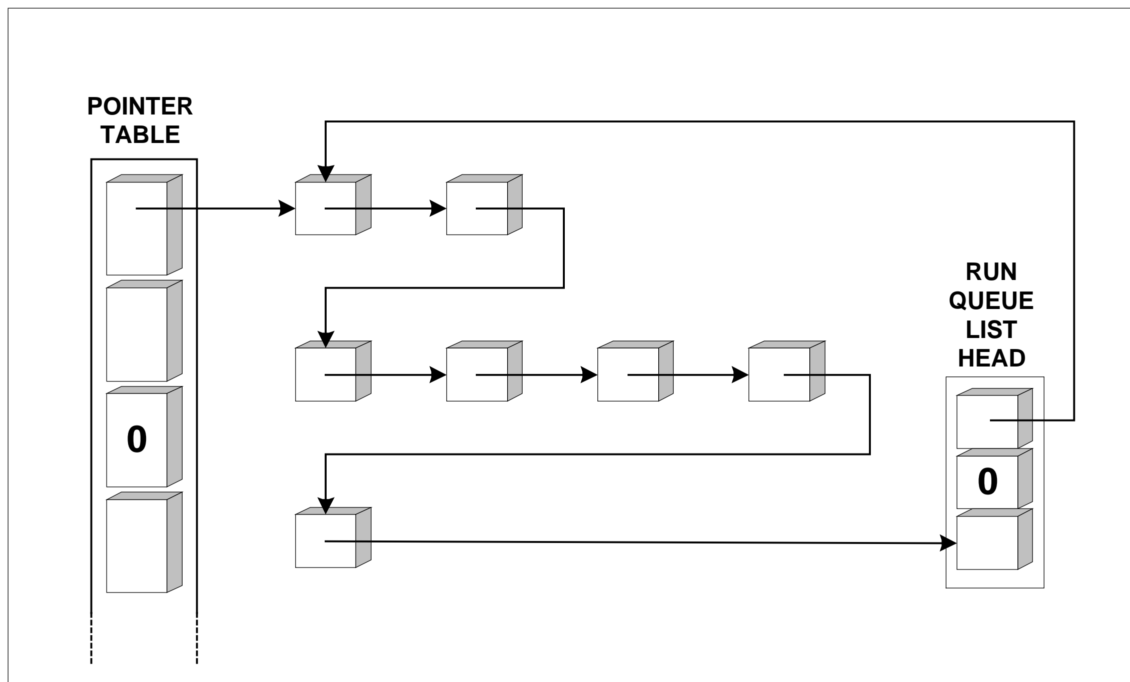


**Diagram 6: Run Queue and Shortcut Pointer Table**

See the later section on “Scheduling Examples” for some examples of the use and modification of this table.

For each priority level, the "bit table" contains 1 bit, which indicates whether any processes at that level are in the READY state. The bit table allows for deterministic insertion into the run queue.

The simplicity of the dispatcher allows context switches to take place very quickly. The complex task of implementing scheduling policies is transferred to higher level routines which can be adapted to suit the application.

## 5.3 Scheduling Policy

The scheduling policy of a system is the part of the scheduler which calculates the appropriate order for processes within the run queue, and selects the process to be run next.

Only processes in the READY state are allowed a place in the run queue. When a process is moved into the READY state, a call is automatically made to the function that governs scheduling policy, so that the process's place in the run queue may be established.

This function is *sys/kn/sched/schedule*. It is usually called by *sys/kn/proc/resume*, *sys/kn/proc/wake*, *sys/kn/proc/deschedule*, or another system function responsible for moving the relevant process into the READY state. It may not be called from user code. After performing any necessary calculations, *sys/kn/proc/schedule* links the process into the correct position in the run queue.

The way in which the processes are ordered is based on various parameters. These parameters may be assigned at design time or run-time, depending upon the nature of the policy in question. The priority in Rate Monotonic scheduling, for example, must be statically assigned at the system design

phase. The laxity in Minimum Laxity First scheduling and Maximum Urgency First scheduling, and the 'time to deadline' in Earliest Deadline First scheduling, however, must be re-calculated and modified dynamically during the system operation.

int**e**nt supports the use of a wide variety of scheduling policies and scheduling models, a few of which are listed below.

- Minimum Laxity First. (MLF)
  This is a scheduling method which orders tasks on the basis of their 'laxity.' Laxity can be defined as the gap between the current time, and the latest time at which the process could start and still complete before its deadline passes. The task with the lowest laxity value is dispatched first.

- Maximum Urgency First. (MUF)
  This is a modification of the MLF scheduling method. Here, tasks are divided among several priority levels. Tasks with higher priorities are always dispatched first. Within a priority level, the task with the smallest laxity is dispatched first.

  This modification makes it possible to control which tasks fail their deadlines in a transient system overload. When using policies like EDF and MLF, it is not possible to control which tasks fail.

- Earliest Deadline First. (EDF)
  This is a scheduling method which orders tasks on the dispatch list according to their deadlines. The task with the closest deadline is dispatched first.

At present, int**e**nt uses by default a Fixed Priority scheduling mechanism, whereby a process is assigned a priority on some basis, and scheduling is carried out based entirely on the relative priorities of the processes. These priorities are statically assigned as part of system analysis, but can also be altered by the user during run-time.

If there are two or more processes at a particular priority level, the one which was started first will run first, until it no longer wishes to run, followed by the next “oldest” and so on, providing timeslicing is switched off. If timeslicing is switched on, it is slightly harder to predict in which order the processes will run.

Examples of static analysis methods are:

- Rate Monotonic Scheduling. (RM)
  This method assigns task priorities for a real time system solely on the basis of the period of the tasks. The task with the shortest period (the highest frequency) has the highest priority. Since all the calculation for this method is completed at design time, and the policy is fully static at run-time, the scheduler policy unit is very simple.

  In theory, using this method, it should be possible to calculate the completion times of all periodic tasks, and in mathematical terms to guarantee that none will fail their deadlines. However, the CPU utilisation must be significantly lower than 100% for this guarantee to hold.

  For a particular set of tasks with known worst case execution times, etc, it is possible to calculate the maximum CPU utilisation that can occur without compromising this guarantee of schedulability. If this threshold is exceeded, this may result in a temporary overload, and some tasks may fail to meet their deadlines. The order in which tasks will fail to meet their deadlines is always lowest priority first.

  There are numerous extensions to the method to provide for aperiodic tasks and dynamic priority modification.

- Deadline Monotonic Scheduling. (DMS)
  This method assigns task priorities for a real time system, based solely on task deadlines. The task with the shortest interval between triggering and deadline is given the highest priority.

## 5.4 Using more than one scheduling policy within an intent system

Many different scheduling policies can operate simultaneously in a single intent system. As in other operating systems that share this feature, each scheduling policy running within an intent system must have exclusive rights to operate upon a range of priority levels. There can be no overlap between the priority ranges governed by different scheduling policies, since their modes of calculation may be incompatible.

The EDF scheduling model, for example, uses no real concept of "priority," only of execution order. For this reason it requires only one priority level, and may operate by modifying the order of the processes within that priority level. When a process is moved into the READY state, its deadline is calculated and it is positioned in the run queue so that its deadline is later than that of the previous node in the list, and earlier than that of the next node in the list.

In the case of Rate Monotonic Analysis, on the other hand, the priorities of processes are calculated statically at design time. It is therefore impossible to determine whether an EDF process with a particular deadline should be executed before a process given a priority through RM, unless the EDF scheduler schedules its tasks at one priority level, and the tasks assigned by RM analysis are scheduled at a different range of priorities.

Although the necessity of segregating different scheduling models can be seen as a limitation, there are also advantages to maintaining separately scheduled task sets. The partitioning system can, for example, help to control the failure of critical tasks in the case of a period of transient overload.

If a particular set of tasks exceed their allotted maximum CPU utilisation, the processor may overload for a brief interval, causing some of the tasks to fail to meet their deadlines. Scheduling mechanisms such as EDF and MLF have the disadvantage that in such cases it is impossible to ensure that those tasks that fail are non-critical.

By segregating the different scheduling policies, it is possible to partition the processes in the system into critical and non-critical tasks, or even to divide them into a hierarchy of different levels of importance. This makes it possible to predict which processes will fail to meet their deadlines during transient overload situations.

Below is an example of a system designed to accommodate numerous scheduling models, including EDF and RM mechanisms.

### 5.4.1 Scheduling Example

A system designer has chosen that a set of processes with their priorities statically assigned using RM analysis should have higher priorities than all other processes in the system. The scheduling guarantees provided by using Rate Monotonic analysis are therefore not affected by the presence of other processes and other scheduling methods in the system.

For this example, the priority levels are partitioned as shown in the table below.

| Scheduling Policy | Priority Range |
|---|---|
| Statically Assigned Priorities by RM Analysis | 0-15 |
| EDF Scheduling | 16 |
| EDF Scheduling | 17 |
| Fixed Priority (Timesliced) | 18-255 |

When a process is started, its priority is specified by its parent, and it is thus assigned to a particular scheduling mechanism. It is possible for a process to change its priority and even its scheduling mechanism using the *kn_proc_setparams* function, but the system designer should be careful not to allow processes to change their scheduling mechanism unintentionally.

Note that the technique used by the fixed priority scheduler has a bounded execution time with a very small jitter. A jitter is the random variation in the timing of a signal. In this case, the jitter is slightly smaller if there is another process in the priority level.

The technique used by the EDF and MLF schedulers has a bounded execution time if the number of processes being scheduled by that technique is known. The jitter is quite large, which is a result of the search required by EDF and MLF schedulers.

## 5.5 Deadline Failures

Deadline failure detection may be provided for any process which provides the system with its deadline, and indicates when it has completed its tasks. The process usually indicates this by ceasing to execute.

If the process also provides its best case execution time, the system may compare this to the time to deadline, and thus detect some deadline failures before the process executes. This enables the process to take appropriate action. The process may switch to a mode requiring less calculation, increasing the likelihood of completion within deadline. Alternatively, other processes that would otherwise have failed their deadlines may be allowed to execute in its place, and perhaps complete successfully.

In the case of a deadline failure, a SIGDLINE signal is generated for the process. The system will ignore this signal, unless the user has set up a handler for this signal using the *sys/kn/sig/action* function.

Typically, upon receipt of a SIGDLINE signal, the deadline failure signal handler would set a flag indicating a deadline failure. This would allow the failed process to modify its operation, so as to make future deadline failures less likely. Another common action of the handler would be to use the *lib/longjmp* function and cause the program to change to an alternate code path. For example, in a polygon drawing tool, it may be useful to set up the system to provide a deadline failure exception after a certain period of time, so that the amount of detail in the scene could be reduced for the next frame.

In high level languages that support deadline exception handling, such as Java™, the following type of construct can be used to catch deadline exceptions:

```
try {
 /* Execute some code which may fail to meet its deadline */
 } catch (DeadlineException d) {
 /* Handle the exception here */
 }
```

In high level languages that do not have built-in exception handling support, the following type of functionality will be available:

```
for(;;) {
 sleep(-1); /*Wait until triggered by interrupt or something similar */
 periodic_start (gettime()+period, rt_function, rt_exception);
 }
```

```
int rt_function() {
/* Execute some code which may fail to meet the deadline */
}
int rt_exception() {
/* Handle the deadline exception here */
}
```

Both of these mechanisms will be implemented using int**e**nt’s low-level exception/signal handling mechanism. Further details of int**e**nt’s support for signals may be found later in this document in the appropriate section.

# 6. Process Synchronisation

intent provides several methods of process synchronisation. These include counting semaphores, mutexes, event flags, and mailboxes.

## 6.1 Counting Semaphores

Counting semaphores are structures in memory. These structures provide access protection for a resource to which a limited number of simultaneous accesses are permitted, and ensure that the set limit is not exceeded.

A process can only access a semaphore if it knows its address, or can find its address (through communication with a parent process, for example).

The counting semaphore maintains a count of the 'vacancies' available for additional processes to access the relevant resource. The *wait*, *timedwait* and *trywait* operations decrement the count, and the *post* operation increments the count. If the count falls to zero, there are no more vacancies, and the next process to call one of the **wait* functions on the semaphore is blocked until a process calls *post* on the semaphore. Multiple processes may be blocked on the semaphore at the same time, in which case they are queued. The queue is ordered either by priority or on a first-in-first-out (FIFO) basis, depending on the flags with which the semaphore was created.

Semaphores in intent do not support any priority-inversion protection mechanisms (see Glossary). Most priority-inversion protection mechanisms act by modifying the priority of the 'owner' of the protected resource. Since a semaphore does not have an 'owner', unlike a mutex, it is difficult to protect in this manner. Many processes can successfully wait on a semaphore at the same time, unless the count is one, in which case the semaphore effectively acts as a mutex (see below).

intent semaphores provide the same functionality as POSIX.4 unnamed (memory) semaphores, with the following modifications and improvements:

- The posix "pshared" parameter on *sem_init()* does not exist in the intent function *sys/kn/sem/init*.
- The intent function *sys/kn/sem/init* has an extra parameter "flags" which the posix function lacks.
- intent semaphores provide an extra function *sys/kn/sem/timedwait*.
- intent semaphores provide an extra function *sys/kn/sem/trymwait*.
- intent semaphores provide an extra function *sys/kn/int/sem/post*, which can be used to post semaphores from hardware interrupt handlers and intent timers.

The function mapping between posix semaphores and intent semaphores is shown in Table 2 - Correspondence between intent and Posix.4 semaphore functions.

| **intent Function** | **Posix.4 Function** |
|---|---|
| `int kn_sem_init(ELATE_SEMAPHORE *sem, unsigned int count, int flags);` | `int sem_init(sem_t *sem, int pshared, unsigned int count);` |
| `int kn_sem_destroy(ELATE_SEMAPHORE *sem);` | `int sem_destroy (sem_t *sem);` |
| `int kn_sem_trywait(ELATE_SEMAPHORE *sem);` | `int sem_trywait(sem_t *sem);` |
| `int kn_sem_wait(ELATE_SEMAPHORE *sem);` | `int sem_wait(sem_t *sem);` |
| `int kn_sem_timedwait(ELATE_SEMAPHORE *sem, time_t timeout);` | No equivalent |
| `int kn_sem_post(ELATE_SEMAPHORE *sem);` | `int sem_post(sem_t *sem);` |
| `int kn_sem_getvalue(ELATE_SEMAPHORE *sem);` | `int sem_getvalue(sem_t *sem, int *value);` |

| | |
|---|---|
| `int kn_int_sem_post(ELATE_SEMAPHORE *sem);` | No equivalent |
| `int kn_int_sem_trymwait(ELATE_SEMAPHORE *sem, int num);` | No equivalent |

### 6.1.1 Initialise a Semaphore - sys/kn/sem/init

This function is used to initialise the semaphore structure to which the pointer parameter points. The semaphore count is initialised to the value specified, which must be greater than or equal to zero.

The flags parameter controls the order in which blocked processes are unblocked. Exactly one of the following flag bits must be set in the flags parameter.

- SEM_FIFO : This flag specifies that when processes block waiting for the semaphore, they should be unblocked in a first-in-first-out order.
- SEM_PRIO : This flag specifies that when processes block waiting for the semaphore, they should be unblocked in a highest priority first order. If multiple processes of the same priority are blocked on the same semaphore, they are unblocked in a first-in-first-out order.

### 6.1.2 Destroy an Unnamed Semaphore - sys/kn/sem/destroy

This function is used to destroy the specified semaphore. Only a semaphore which was initialised using *sys/kn/sem/init* may be destroyed using this function. Calling *sys/kn/sem/destroy* does not free any memory associated with the semaphore.

The effect of subsequent use of the specified semaphore is undefined, unless it has been re-initialised by another call to *sys/kn/sem/init*.

Destroying a semaphore that has processes waiting on it has the effect of unblocking all of those processes and returning the error code EINVAL to them.

### 6.1.3 Wait on a Semaphore - sys/kn/sem/wait

This function decrements the count of the specified semaphore. If the resulting count is greater than or equal to zero, 0 is returned, to indicate successful completion. If the resulting count is negative, the calling process blocks until another process calls *sys/kn/sem/post* or the semaphore is destroyed using *sys/kn/sem/destroy*.

### 6.1.4 Wait on a Semaphore, non blocking - sys/kn/sem/trywait

The *sys/kn/sem/trywait* function decrements the count associated with the semaphore, and returns successfully if the value of the count is greater than zero. If the count is not greater than zero it returns EBUSY to indicate that no 'vacancies' are available at the semaphore.

### 6.1.5 Decrement the semaphore count by the specified amount without blocking - sys/kn/sem/trymwait

The *sys/kn/sem/trymwait* function attempts to decrement the semaphore count by the specified amount. If the amount specified is greater than the semaphore count, the semaphore count is set to zero and the value returned is the amount by which the count was decremented.

The input value is treated as an unsigned value.

### 6.1.6 Wait on a Semaphore with timeout - sys/kn/sem/timedwait

This function decrements the count of the specified semaphore. If the resulting count is greater than or equal to zero, 0 is returned, to indicate successful completion. If the resulting count is negative, the calling process blocks until another process calls *sys/kn/sem/post*, the semaphore is destroyed using *sys/kn/sem/destroy* or the amount of time specified by the timeout parameter has passed.

### 6.1.7 Post to a Semaphore - sys/kn/sem/post

If there are any processes waiting for the specified semaphore, the 'first' of them is unblocked. This process will be at the head of the queue which is ordered in accordance with the SEM_PRIO or SEM_FIFO flag specified when the semaphore was initialised. If there are no processes waiting for the semaphore, its count is incremented.

### 6.1.8 Get the Value of a Semaphore - sys/kn/sem/getvalue

This function returns the value of the specified semaphore's count. The state of the semaphore is not affected.

If the value returned is positive, this indicates the number of 'vacancies' at the semaphore. If the semaphore is locked, the value returned is a negative number whose absolute value represents the number of processes waiting for the semaphore.

The value returned indicates the semaphore count, or the number of waiting processes, at one point during the *sys/kn/sem/getvalue* call. It does not necessarily represent the state of the semaphore at the time the function returns.

## 6.2 Mutexes

A mutex (mutual exclusion object) serves a slightly different purpose from a semaphore. Whereas a semaphore maintains a limit to the number of processes achieving access to a resource, a mutex provides mutual exclusion, like a counting semaphore with a count of one.

Mutexes are often used to provide exclusive access to data structures that must be accessed by no more than one piece of code at a time. It is sometimes highly undesirable for two processes to operate upon the same data structure at once. For instance, if two processes attempted to read, modify and write a memory variable simultaneously, there would a danger of losing one of the changes.

A mutex may only be locked by one process at a time, and in this way provides mutual exclusion. While it is locked, any other process attempting to access it will block until it next becomes unlocked. In intent, mutexes allow substantial control over the way in which processes block. A number of mechanisms are provided to make sure that a high priority process is forced to wait as little time as possible for any mutex locked by a lower priority process.

There are several ways to ensure mutual exclusion. Interrupts and facilities for pre-emption, for example, may be disabled to prevent task switches. However, these tactics seriously delay and hamper the system's ability to respond to interrupts, to reschedule tasks, etc. Unconnected processes are prevented from running, even if they do not attempt to access the relevant data structure. There is a danger of priority inversion, since a high priority process could be blocked indefinitely by a process with a lower priority.

A mutex does not impede unrelated tasks in the same fashion. Even related tasks can still execute providing they do not attempt to access the resource protected by the mutex. In addition, mutexes are still reliable when operating within a shared memory multiprocessor system, whereas disabling

interrupts on one chip in such a system does not necessarily prevent access to the data structure in question from other processors.

A mutex also has several advantages over a semaphore with a count of one. Semaphores have no 'owner'. Any process with access to the semaphore can 'post' it, allowing a blocked process access to the resource. There is no way to ensure that the semaphore can only be posted by its current 'owner.' The lack of an 'owner', furthermore, prevents the implementation of a mechanism to prevent priority inversion.

Mutexes are not restricted in this way. Once a process locks a mutex, it becomes the 'owner,' and no other process can unlock it. Since the mutex contains a record of its 'owner,' furthermore, it can be subjected to techniques such as the Basic Priority Inheritance Protocol or the Highest Locker Protocol (see glossary) which are designed to avoid priority inversion.

### 6.2.1 Initialise Mutex - sys/kn/mtx/init

This function initialises a mutex structure, making it ready for use. If this function succeeds, the specified mutex can be used as a parameter to the functions *sys/kn/mtx/destroy*, *sys/kn/mtx/lock*, *sys/kn/mtx/trylock*, *sys/kn/mtx/timedlock*, *sys/kn/mtx/unlock* and *sys/kn/mtx/ islocked*. This tool must not be called on a mutex which is already initialised under any circumstances.

The flags parameter specifies the order in which processes are released from the blocked state, if there are multiple processes waiting for the mutex at one time. It also specifies the method used to avoid priority inversion. The valid values are shown below. Exactly one of MTX_FIFO and MTX_PRIO should be specified, and exactly one from MTX_BPIP, MTX_HLP, MTX_SIG and MTX_NONE should be specified. In addition, MTX_SIGMASK may be specified in combination with any of the other flags except MTX_SIG. MTX_LOCK and MTX_RECURSIVE may be specified in combination with any other flags.

- MTX_FIFO :
  This flag specifies that when multiple processes block waiting for the mutex, they should be unblocked in a first-in-first-out order.
- MTX_PRIO :
  This flag specifies that when multiple processes block waiting for the mutex, they should be unblocked in a highest priority first order. If multiple processes of the same priority are blocked on the same semaphore, they are unblocked in a first-in-first-out order.
- MTX_BPIP :
  This flag specifies that the Basic Priority Inheritance Protocol should be used to avoid priority inversion. The "ceiling" parameter is unused.
- MTX_HLP :
  This flag specifies that the Highest Locker Protocol should be used to avoid priority inversion. The parameter "ceiling" specifies the priority ceiling of the mutex.
- MTX_NONE :
  This flag specifies that there is no mechanism used to avoid priority inversion. The "ceiling" parameter is unused.
- MTX_SIG :
  This flag specifies that priority inversion is to be avoided by using the priority signalling mechanism. The "ceiling" parameter is unused.
- MTX_SIGMASK :
  This flag specifies that when a process locks the mutex, its signal disable flag should be set. This prevents the process from receiving any signals while it is in control of the mutex. The process's signal disable flag returns to its original state when the mutex is unlocked. The MTX_SIGMASK flag is usually set to prevent a process *longjmp*ing out of its signal handler without relinquishing the mutex lock. This flag also prevents high-level-language exceptions from being processed.
- MTX_LOCK:

This flag specifies that the mutex should be created in a locked state. If this flag is specified, the mutex can never be available to any process other than the caller, unless first this call returns and the calling process explicitly calls *sys/kn/mtx/unlock*.

- MTX_RECURSIVE:
  This flag specifies that recursive locking of the mutex should be allowed. This means that *sys/kn/mtx/lock* will never return EDEADLK. Instead, attempts by the 'owner' to lock the mutex recursively will succeed, and a count will be maintained such that the mutex must be unlocked the same number of times as it has been locked before the mutex is released. Processes other than the owner of the mutex will block, or return an error code, as normal.

See the glossary for descriptions of the Highest Locker Protocol, Basic Priority Inheritance Protocol, Priority Signalling and priority inversion.

Upon successful initialisation, the mutex is in an unlocked state unless the MTX_LOCK flag was specified.

### 6.2.2 Destroy Mutex - sys/kn/mtx/destroy

This function destroys the specified mutex, in effect deinitialising it. It does not free any memory associated with the mutex. A destroyed mutex can be re-initialised by calling sys/*kn/mtx/init*. The effects of using a destroyed mutex are undefined.

A mutex may only be destroyed while it is unlocked, or while it is locked by the calling process. In the latter case, any blocked processes are unblocked, and return EINVAL, and the lock is automatically released.

Attempting to destroy a mutex which is locked by another process will fail.

### 6.2.3 Lock Mutex - sys/kn/mtx/lock

If the specified mutex is unlocked, the *sys/kn/mtx/lock* function locks it and returns. The calling process thus becomes the 'owner' of the mutex.

If the mutex is already locked by the calling process, the action is dependent on whether the mutex was initialised with the MTX_RECURSIVE flag. If it was not, EDEADLK is returned and no further action is taken. If the flag was set, this function returns 0 (success) and a count is incremented to indicate the number of recursive locks which the process has on the mutex. The function *sys/kn/mtx/unlock* must be called the same number of times in order to release the mutex.

If the mutex is locked by another process, the calling process is placed on the queue of processes waiting for the mutex lock. Any priority inversion prevention measures specified at mutex initialisation time are automatically applied.

If an unmasked signal becomes pending, but signals were disabled before entry to the function, then EINTR is returned. Internally, this function will retry the blocking operation if it can to ensure that the signal has been properly processed. It should be noted that in the case of an unmasked signal becoming pending, EINTR will only be returned if this signal processing has been disabled outside of the scope of this function. If this is unacceptable then *sys/kn/mtx/siglock* should be called instead.

### 6.2.4 Lock Mutex, non blocking - sys/kn/mtx/trylock

If the specified mutex is unlocked, the *sys/kn/mtx/trylock* function locks it and returns zero.

If the mutex is already locked by the calling process, the action is dependent on whether the mutex was initialised with the MTX_RECURSIVE flag. If it was not, EDEADLK is returned and no further

action is taken. If the flag was set, this function returns successfully, and a count is incremented to indicate the number of recursive locks which the process has on the mutex. The function *sys/kn/mtx/unlock* must be called the same number of times in order to release the mutex.

If the mutex is locked by another process, the function returns an error code indicating the error.

### 6.2.5 Lock Mutex with timeout - sys/kn/mtx/timedlock

If the specified mutex is unlocked, the *sys/kn/mtx/timedlock* function locks it and returns.

If the mutex is already locked by the calling process, the action is dependent on whether the mutex was initialised with the MTX_RECURSIVE flag. If not, EDEADLK is returned and no further action is taken. If the flag was set, this function returns successfully, and a count is incremented to indicate the number of recursive locks which the process has on the mutex. The function *sys/kn/mtx/unlock* must be called the same number of times in order to release the mutex.

If the mutex is locked by another process, the calling process is placed on the queue of processes waiting for the mutex lock, as specified at the initialisation of the mutex. If the amount of time specified by the timeout parameter passes before the mutex becomes available, this function returns ETIMEDOUT.

If an unmasked signal becomes pending, but signals were disabled before entry to the function, then EINTR is returned. Internally, this function will retry the blocking operation if it can ensure that the signal has been properly processed. If an unmasked signal becomes pending, EINTR will only be returned if this signal processing has been disabled outside of the scope of this function.

### 6.2.6 Unlock Mutex - sys/kn/mtx/unlock

This function is called by the current owner of the mutex to unlock it. If the MTX_RECURSIVE flag was specified when the mutex was initialised, this operation will decrement the count of recursive locks held by the calling process. If this decremented count is non-zero, the function simply returns a value of 1 without taking any further action. When this count reaches zero, the mutex is unlocked normally, as described below.

If this function is called by a process which is not the owner of the mutex, the unlock operation fails. Calling *sys/kn/mtx/unlock* when the mutex is not locked results in undefined behaviour.

If there are processes blocked on the mutex when it becomes available, one of the processes is unblocked and becomes the 'owner' of the mutex, and 0 is returned. If multiple processes are blocked, the decision as to which is rescheduled is based on the flags parameter with which the mutex was initialised.

If the recursive lock count has dropped to 0, then the mutex is fully unlocked and the return value is 0.

### 6.2.7 Locks a mutex, retrying if interrupted by a signal - sys/kn/mtx/siglock

This tool is a variant of sys/kn/mtx/lock. The difference is that this tool will never return EINTR. Instead, if an unmasked signal occurs which would otherwise have resulted in a return value of EINTR, the locking attempt will be retried until EINTR is no longer returned.

### 6.2.8 Attempt to lock a mutex with timeout, retrying if interrupted by a signal - sys/kn/mtx/sigtimedlock

This tool is a variant of *sys/kn/mtx/timedlock*, but differs in that this tool will never return EINTR. In the case of an unmasked signal which would have caused *sys/kn/mtx/timedlock* to return a value of EINTR, this tool retries the locking attempt until EINTR is no longer returned.

### 6.2.9 Return lock status of mutex - sys/kn/mtx/islocked

This function returns the mutex lock status, indicating whether the mutex is locked or unlocked at that time. The state of the mutex is not modified by this function.

## 6.3 Event Flags

More versatile than either semaphores or mutexes, event flags allow multiple tasks to wait for multiple events.

An event flag represents 32 events which can be waited on in combination. Each event is represented by a bit in the event flag pattern; if the relevant bit is not set, then the event has not yet happened, if set then it has. The meaning of these events is not predetermined, and is entirely defined by the application using the flag. Each task can separately specify the set of events of which it needs notification, and the combination of events for which it is waiting. A single event trigger operation can trigger multiple tasks.

### 6.3.1 Initialise an event flag structure - sys/kn/evf/init

This function initialises the event flag structure, and sets it to the specified pattern. The meaning of the bits in the flag pattern is not passed in; it is the responsibility of the application programmer to ensure that the correct bits in the pattern are set when necessary.

### 6.3.2 Destroy an event flag - sys/kn/evf/destroy

This function destroys the specified event flag, rendering it unusable. Any attempt to use this event flag after its destruction gives undefined results. This function does not free the memory associated with the event flag structure.

If there are processes waiting for the event flag, they are restarted. In each case the function that the process called in order to wait upon the event flag (*sys/kn/evf/wait*, *sys/kn/evf/trywait* or *sys/kn/evf/timedwait*) returns zero.

### 6.3.3 Set the flag pattern of an event flag - sys/kn/evf/set

This function sets the event flag pattern, passing in a bit pattern to OR into the current event flag pattern. Any processes waiting upon the event flag will be unblocked if their conditions are met by the new flag pattern of the event flag.

An 'interrupt-safe' version of this tool, *sys/kn/int/evf/set*, is also available. *sys/kn/int/evf/set* can safely be called from inside an interrupt. Further details are available in the later section on “Interrupt Handling.”

### 6.3.4 Clear the flag pattern of an event flag - sys/kn/evf/clr

This function sets the event flag pattern to the logical AND of its value parameter with the current event flag pattern. Processes are not unblocked due to this operation.

### 6.3.5 Waiting on an Eventflag

Processes may call one of three different functions in order to test an event flag for a condition. These are *sys/kn/evf/wait*, *sys/kn/evf/trywait*, and *sys/kn/evf/timedwait*. Each of these takes a pointer to the event flag to be tested, an integer representing the flag pattern against which the existing flag is to be compared, and a mode parameter indicating the manner in which the two flag patterns are to be compared. The *sys/kn/evf/timedwait* function also takes a long integer which specifies the timeout period in nanoseconds.

The mode parameter may give one of two values, indicating either the EVFF_AND mode or the EVFF_OR mode.

- EVFF_AND
  If this flag is specified, the comparison will only evaluate to TRUE if all of the bits set in the pattern parameter are set in the event flag's pattern.
- EVFF_OR
  If this flag is specified, the comparison will evaluate to TRUE if any of the bits set in the pattern parameter are set in the event flag's pattern.

In addition, the EVFF_CLR flag may optionally be set in the mode parameter.

- EVFF_CLR
  If this flag is specified, all bits of the event flag will be cleared to zero after the wait conditions of the waiting task are satisfied.

### 6.3.6 Wait on event flag until a specific condition is fulfilled - sys/kn/evf/wait

The *sys/kn/evf/wait* function is called by a process to test the event flag for the specified condition. The function blocks if the condition is not TRUE. If a call to *sys/kn/evf/set* is later employed to change the flag pattern of the event flag, the condition will be retested, and this call will return if the new pattern matches the specified values.

This function returns the value of the event flag pattern when the specified condition is satisfied. If the EVFF_CLR flag is specified, the function returns the value of the event flag before the flag pattern is cleared to 0.

The value 0 may not be specified in the input pattern parameter, since it satisfies no wait conditions. Consequently a return of 0 is always an error, and may not be mistaken for a record of the flag pattern.

When multiple processes are waiting on an event flag, a single call to the *sys/kn/evf/set* function may satisfy the wait conditions of multiple processes. If a process which has specified the EVFF_CLR flag is released in this manner, the event flag pattern is cleared to 0 as the process is released. Any processes behind this process in the queue are not released.

### 6.3.7 Test event flag for specified event flag pattern, non blocking - sys/kn/evf/trywait

The *sys/kn/evf/trywait* function tests the event flag for the specified condition. Instead of blocking until the condition is met, it immediately returns the logical result of this test (TRUE or FALSE).

The value 0 may not be specified in the input pattern parameter, since it satisfies no wait conditions.

### 6.3.8 Wait on event flag for specific condition (blocking), with timeout - sys/kn/evf/timedwait

The *sys/kn/evf/timedwait* function is called by a process to test for the specified condition. If the condition is not TRUE, the function blocks. It may be restarted by a subsequent call to *sys/kn/evf/set* with a matching pattern, or by the expiration of the specified timeout period.

This function returns the value of the event flag pattern when the specified condition is satisfied. If the EVFF_CLR flag is specified, the function returns the value of the event flag before the flag pattern is cleared to 0.

The value 0 may not be specified in the input pattern parameter, since it satisfies no wait conditions. Consequently a return of 0 is always an error, and may not be mistaken for a record of the flag pattern.

When multiple processes are waiting on an event flag, a single call to the *sys/kn/evf/set* function may satisfy the wait conditions of multiple processes. If a process which has specified the EVFF_CLR flag is released in this manner, the event flag pattern is cleared to 0 as the process is released. Any processes behind this process in the queue are not released.

### 6.3.9 Get event flag information - sys/kn/evf/info

This function returns information about the specified event flag by filling in the supplied Event Flag Information Structure, defined in the section on “Data structure definitions.”

The field holding information about the event flag pattern contains the pattern at the time that the structure is filled. This need not be the event flag pattern by the time this function returns. The field giving information about the processes waiting on the flag contains 0 if there are no processes waiting on the event flag, or a non-zero value if there are one or more processes waiting.

## 6.4 Reader/Writer Locks

Sometimes when a resource is protected from simultaneous use by two or more processes, this places it under unnecessary restriction. In particular, for many resources it is acceptable for multiple processes to simultaneously use the data structures involved provided that no process attempts to concurrently modify these structures. Reader/writer locks are a locking mechanism designed to permit this.

A reader/writer lock may be locked by a process in one of two modes. If it is locked to ‘read’ then other processes may still obtain the lock in ‘read’ mode. If it is locked to ‘write’ then no other process can obtain the lock in either mode.

In addition, if one or more processes are blocked on the lock waiting to write then no new processes may acquire the lock in read mode.

### 6.4.1 Initialises a reader/writer lock - sys/kn/rwlock/init

This tool initialises a reader/writer lock structure. The function is passed a pointer which must point to a block of memory RWLOCK_SIZE bytes in length (this constant is defined in the document *sys/kn/rwlock/rwlock.inc*), or `sizeof(ELATE_RWLOCK)` if programming in C. The returned parameter contains an error code, or 0 if the function has completed successfully.

### 6.4.2 Destroys a reader/writer lock - sys/kn/rwlock/destroy

This function is used to destroy the specified reader/writer lock. The tool is passed a pointer to the relevant reader/writer lock, and returns an error code, or 0 if it completes successfully. Only a reader/writer lock which was initialised using *sys/kn/rwlock/init* may be destroyed using this function.

### 6.4.3 Waits on a reader/writer lock (blocking, no timeout) - sys/kn/rwlock/wait

This tool attempts to lock the reader/writer lock. The function is passed a pointer to the relevant reader/writer lock, and an integer which indicates the mode of lock. If the latter parameter is 1 then the tool attempts to acquire the lock in 'write' mode, otherwise 'read' mode is selected. In either case, this tool blocks until the lock is successfully acquired.

### 6.4.4 Waits on a reader/writer lock (non-blocking) - sys/kn/rwlock/trywait

This tool behaves similarly to *sys/kn/rwlock/wait* except that the tool simply returns an EBUSY error code if the lock cannot immediately be acquired. This tool never blocks.

### 6.4.5 Waits on a reader/writer lock (blocking, with timeout) - sys/kn/rwlock/timedwait

This tool behaves similarly to *sys/kn/rwlock/wait* except that it takes an additional input parameter, which gives a timeout period in nanoseconds, or a value of –1 for an indefinite wait. After the period of time specified by this parameter has expired, the tool ceases to wait and returns the error value ETIMEDOUT.

### 6.4.6 Unlocks a reader/writer lock - sys/kn/rwlock/unlock

This tool unlocks a lock reader/writer lock. The function is passed a pointer to the relevant reader/writer lock, and returns either an error code or 0 if it has completed successfully. It should be noted that if the lock is in read mode it must be unlocked before any attempt to acquire it in write mode or use *sys/kn/rwlock/upgrade*.

### 6.4.7 When holding the lock as a reader, become a writer - sys/kn/rwlock/upgrade

This tool attempts to acquire a reader/writer lock in write mode for a caller who already holds it in read mode. There are three possible outcomes.

If the call is successful then 0 is returned. This guarantees that no other process has obtained the lock for writing in the meantime.

If the return value is EBUSY this means that another process is entitled to obtain the write lock before the caller. This may happen if, for example, there is a prior call to *sys/kn/rwlock/upgrade* by another process. Alternatively, this other process may have called *sys/kn/rwlock/*wait*, passing in 1 as the lock mode parameter so that the resource is held in write mode. In such a case, when *sys/kn/rwlock/upgrade* returns the resource is not locked by the caller, not even in read mode. This is because the lock must be unlocked for the other process to acquire it in write mode.

The third possible outcome is a return value of EAGAIN. In this case, another process succeeded in acquiring the lock in write mode before the caller, but the caller now has the lock in write mode.

These error conditions must not be treated as identical or the lock will become corrupted.

## 6.5 Mailboxes

Since early in the development of intent, mailboxes have been a core means of effecting communication and synchronisation between processes and other objects within the system.

intent's base functionality for message passing is asynchronous, allowing processes to continue with other tasks after sending messages instead of lying idle while waiting for replies. However, the asynchronous functions may also be used to achieve synchronous message passing. A synchronous message passing model can be implemented using intent mailboxes by having the process send the message block until a reply is received.

Unlike mutexes, semaphores and event flags, which may occupy any area of memory, mailboxes are created by the intent system. When a mailbox is created, the intent system returns a "Mailbox ID," a 64-bit quantity which is subsequently used to refer to that particular mailbox. The ID is used when sending or reading mail from a mailbox, or when freeing a mailbox that is no longer required.

The mailbox ID is unique across the entire intent processor network. Processes on different processors, therefore, can use the ID to communicate, without the different applications involved requiring any special code to handle this.

When a message arrives at its destination mailbox, it is placed in a queue of the other messages waiting to be read. Any number of messages may be queued in a mailbox simultaneously. If there are no messages available to be read from a mailbox, a process may block until a message arrives.

Mailboxes provide the ability to register a message arrival callback function which is called when a message arrives at the mailbox. This function is called after the incoming message has been added to the message list. It may modify the message list in any way, such as by discarding or combining messages. This allows simple implementation of message filters. It also provides functionality such as the automatic combination of several messages into one, which is useful when managing, for example, mouse movement messages in a GUI. The process context in which the message arrival callback function is called depends on several factors, and is described in a later section upon the callback function. A more detailed account of the message arrival callback function is given in the description of *sys/kn/mbox/setcallback*.

*sys/kn/mbox/send* and *sys/kn/int/mbox/send* are the only mailbox functions that may be used on mailboxes on other processors. It is not possible to read from or free a mailbox on a remote processor, since such an implementation would incur significant overheads, but the absence of this feature is seldom a problem.

### 6.5.1 Mail Messages

An intent mail message consists of a header and a body section. Memory for these should be allocated using *sys/kn/mem/allocmail*.

The header structure is described in detail later in this document, in the section on data structures. In short, it contains a list node which allows it to be linked onto a list, the length of the message (including both header and body), and the sender's mailbox ID. The length and sender's mailbox ID should be filled in by the sending process before the mail is sent. The sender ID is only required if a reply is expected. If it is not, the field may be set to zero.

The header is followed by the body of the message. This is comprised of the data to be sent. It may be of any length, and its contents are not interpreted by the system in any way.

The basic intent mailbox system provides facilities which are sufficient for most applications, and which do not add overhead (either memory or execution-time) for application specific features that are

rarely used. More complex behaviour can easily be implemented using the basic mailbox facilities, such as chains of mailboxes or prioritised messages. The latter functionality can be achieved by defining some further structure to a particular applications messages, which can be interpreted by a message arrival callback function to perform priority ordering of the mailbox list or forwarding of the message to another mailbox after processing. It is also possible for different message types to be defined for specific applications.

### 6.5.2 Allocate a mailbox - sys/kn/mbox/alloc

This function allocates and initialises a new mailbox on the same processor as the calling process. The mailbox is initially empty.

This function returns the allocated mailbox ID if successful, otherwise it returns an error code indicating the error.

### 6.5.3 Free a mailbox - sys/kn/mbox/free

This function frees the specified mailbox, and any associated resources. Any messages waiting to be read are discarded. Any processes waiting on the mailbox are woken, and return to their calling contexts with an error code EINVAL.

This function may only be called by a process on the processor upon which the mailbox was allocated.

### 6.5.4 Send Message to mailbox - sys/kn/mbox/send

This function takes a pointer to a mail message, and sends it to the specified mailbox.

Memory for a message should be allocated using *sys/kn/mem/allocmail*. After a call to *sys/kn/mbox/send*, the memory containing the message is considered to belong to the int**e**nt system until it arrives at its destination. No attempt should be made to free or manipulate the memory occupied by the message. Where the message is being sent across a link to a mailbox on another processor, the message memory is automatically freed when it is no longer required. If the destination is a mailbox on the same processor as the sender, the message memory is owned by the process which reads it from the destination mailbox.

If there are any processes blocked on the destination mailbox, the first one in the list is woken from its SLEEP state and receives the message.

If the calling process is running on the same processor as the destination address, any callback function defined for the destination mailbox is called during the execution of *sys/kn/mbox/send*. If they are running on different processors, a callback function defined for the destination mailbox will only execute upon the arrival of the message.

An 'interrupt-safe' version of this tool, *sys/kn/int/mbox/send*, is also available. *sys/kn/int/mbox/send* can safely be called from inside an interrupt. Further details are available in the later section on “Interrupt Handling.”

### 6.5.5 Read mail from mailbox - sys/kn/mbox/read

If there are one or more messages waiting to be read from the mailbox, the first message is removed from the list and returned to the caller.

If no messages are available, the calling process is added to the list of processes waiting to read from the mailbox. This list is ordered by the priority of the waiting processes. When a message arrives, the

first (i.e. highest priority) process on the list is woken and returns to its calling context with a pointer to the mail message.

This function may not be used to read from a mailbox allocated on another processor. It is possible to implement this functionality in user-space, by creating a process on the same processor as the mailbox to service requests from the remote process. However, this functionality is not provided at the system level, and programmers are recommended to avoid this implementation unless absolutely necessary.

If an unblocked signal or a callback becomes pending while a process is blocked in this function, the blocking operation will terminate, and NULL will be returned.

### 6.5.6 Read mail from a mailbox, non-blocking - sys/kn/mbox/tryread

If there are one or more messages waiting to be read from the mailbox, the first message is removed from the list and returned to the caller.

If there are no messages available, a NULL pointer is returned.

This function may not be used to read from a remote mailbox.

### 6.5.7 Read mail from a mailbox, with blocking and timeout – sys/kn/mbox/timedread

If there are one or more messages waiting to be read from the mailbox, the first message is removed from the list and returned to the caller.

If no messages are available, the calling process is added to the list of processes waiting to read from the mailbox. The list order is based upon the priority of the waiting processes. When a message arrives, the first process on the list is woken and returns to its calling context with a pointer to the mail message.

If the amount of time specified in the timeout parameter elapses before the process is woken by the arrival of a message, this function returns NULL.

If an unblocked signal or a callback becomes pending while a process is blocked in this function, the blocking operation will terminate, and NULL will be returned.

This function may not be used to read from a remote mailbox.

### 6.5.8 Set the callback function for the specified mailbox - sys/kn/mbox/setcallback

This function registers a callback function to be called when a message arrives at the mailbox. It should be noted that although this function can be considered to be a callback it is not of the format used by the *sys/kn/callback* functions.

The message arrival callback function will normally be called from the process sending the mail. However, if the message has been sent from an interrupt handler, the callback function is instead called from the next process attempting to read from the mailbox. Alternatively, in the special case of mail being sent to a mailbox on another processor, the callback will be called from an unspecified process.

The input parameter to the message arrival callback function points to the message list of the mailbox. This is a standard int**e**nt double-linked-list, the nodes of which are the messages currently queued in the mailbox. The newly arrived message will be the last on the list.

The message arrival callback function may modify this list by removing messages, or by modifying the data in the messages. If messages are removed from the list in this way, the author of the callback function is responsible for freeing them.

During the execution of the message arrival callback function, the mailbox is locked by the executing process and cannot be modified by any other processes. If the callback function performs any blocking operations, the programmer must be careful to avoid potential deadlock situations.

The message arrival callback function is intended for lightweight operations such as filtering and coalescing, as described above, and not for the actual processing of the messages, it. The chief advantage of this is that messages filtered out or coalesced by the message arrival callback function are never seen by a process reading from the mailbox. Therefore, they do not cause that process to run, and simply discard the message, or duplicate work which could have been performed more efficiently by combining messages. This reduces the number of context-switches, and the overall processing overhead.

### 6.5.9 Call the specified function with the message list of the mailbox - sys/kn/mbox/enumerate

The specified function is called with a pointer to the mailbox's message list, and a user data pointer, allowing operations such as redundant message removal, etc.

The message list parameter to the callback function points to a standard int**e**nt double-linked-list, the nodes of which are the messages currently queued in the mailbox.

The callback function may modify this list by removing messages, or by modifying the data in the messages. If messages are removed from the list in this way, the author of the callback function is responsible for freeing them.

During the execution of the callback function, the mailbox is locked by the executing process and cannot be modified by any other processes. If the callback function performs any blocking operations, the programmer must be careful to avoid potential deadlock situations.

## 6.6 Synchronisation Groups

In a realistic system, it is often necessary to wait for one of several events to occur. These events may be of different types, and may affect different synchronisation objects. For example, it may be necessary to wait on several mailboxes and a semaphore, with a timeout. Some older systems use polling to achieve this. However, since polling forces the system to check the condition of a number of facilities repeatedly, this technique is inefficient and wastes processor resources.

The "synchronisation group" in int**e**nt provides the same functionality without this inefficiency. A "synchronisation group" is a construct that groups synchronisation objects together. A process waiting on this group will receive notification of the first event to occur at any of the grouped synchronisation objects.

When a synchronisation object is associated with a synchronisation group, it should only be accessed as a part of that group, by using the *sys/kn/sgrp/** functions. If it is accessed individually through the normal mutex, semaphore, mailbox or event flag functions, the results may be unpredictable.

### 6.6.1 Initialise Synchronisation Group - sys/kn/sgrp/init

This function initialises the specified synchronisation group structure, making it usable. The synchronisation group initially has no associated synchronisation objects. The maximum number of synchronisation objects which can be associated with the group is specified as a parameter to this function. Attempts to associate more objects than specified on initialisation will fail.

### 6.6.2 Destroy Synchronisation Group - sys/kn/sgrp/destroy

This function destroys the specified synchronisation group structure, making it unusable. The synchronisation group cannot be destroyed while it has associated synchronisation objects. This function does not free any memory associated with the synchronisation group.

Any processes waiting on the synchronisation group are woken, and the EINVAL error code is returned to them.

### 6.6.3 Associate Mutex with Synchronisation Group - sys/kn/sgrp/mtx_assoc

This function associates the specified mutex with the synchronisation group, so that when the mutex is unlocked, the synchronisation group will be notified.

### 6.6.4 Associate Semaphore with Synchronisation Group - sys/kn/sgrp/sem_assoc

This function associates the specified semaphore with the synchronisation group, so that when the semaphore is posted, the synchronisation group will be notified.

### 6.6.5 Associate Mailbox with Synchronisation Group - sys/kn/sgrp/mbox_assoc

This function associates the specified mailbox with the synchronisation group, so that when the mailbox receives mail, the synchronisation group will be notified.

### 6.6.6 Associate Event Flag with Synchronisation Group - sys/kn/sgrp/evf_assoc

This function is slightly different from those listed above, in that it takes not only a pointer to the synchronisation group and a pointer to the event flag, but also a pattern parameter, and a mode parameter. The pattern parameter is needed so that it can be compared to the relevant event flag pattern. The mode dictates the manner in which the two flag patterns are to be compared.

A single event flag may be associated with a synchronisation object multiple times, with different modes and patterns. In such cases, a synchronisation group will be notified if a particular event flag matches any of the specified sets of conditions.

When the event flag matches the specified pattern in the manner decreed by the mode, the synchronisation group is notified.

### 6.6.7 Associate Synchronisation Group with Synchronisation Group - sys/kn/sgrp/sgp_assoc

This function takes as parameters pointers to two synchronisation groups, and associates the second group with the first. Thus, when the second synchronisation group is posted, for any reason, the first synchronisation group will be notified.

### 6.6.8 Disassociate Mutex from Synchronisation Group - sys/kn/sgrp/mtx_disassoc

This function disassociates the specified mutex from the synchronisation group, so that when the mutex is unlocked, the synchronisation group is no longer notified.

### 6.6.9 Disassociate Semaphore from Synchronisation Group - sys/kn/sgrp/sem_disassoc

This function disassociates the specified semaphore from the synchronisation group, so that when the semaphore is posted, the synchronisation group is no longer notified.

### 6.6.10 Disassociate Mailbox from Synchronisation Group - sys/kn/sgrp/mbox_disassoc

This function disassociates the specified mailbox from the synchronisation group, so that when the mailbox receives mail, the synchronisation group is no longer notified.

### 6.6.11 Disassociate EVF Condition from Synchronisation Group - sys/kn/sgrp/evf_disassoc

This function disassociates the specified combination of event flag, pattern and mode from the specified synchronisation object, so that when this pattern and mode combination occur in the specified event flag, the synchronisation object is no longer notified.

If the relevant event flag is associated with the synchronisation group multiple times, only the specified combination of pattern and mode are disassociated from the group. The synchronisation group will still be notified if the event flag satisfies the conditions specified by other combinations of event flag, pattern and mode that are still associated with the group.

### 6.6.12 Disassociate Event Flag from Synchronisation Group - sys/kn/sgrp/evf_destroy

This function disassociates all records which refer to the specified event flag from the specified synchronisation object.

### 6.6.13 Disassociate Synchronisation Group from Synchronisation Group - sys/kn/sgrp/sgp_disassoc

This function takes as parameters pointers to two synchronisation groups, and disassociates the first group from the second. Thus when the second synchronisation group is posted, the first synchronisation group will no longer be notified.

### 6.6.14 Wait on Synchronisation Group – sys/kn/sgrp/wait

If any of the associated synchronisation objects are in the available state (unlocked for a mutex, positive count for a semaphore, etc), information about one of these objects is returned to the caller, and some operation is performed on it to ensure that no other process is triggered by the same object (for details of the structure of the information returned, see below).

If none of the objects is in an available state, the calling process is added to a priority-ordered list of processes waiting for one of the objects in the synchronisation group to become available.

### 6.6.15 Test Synchronisation Group, non-blocking – sys/kn/sgrp/trywait

If any of the associated synchronisation objects are in the available state (unlocked for a mutex, positive count for a semaphore, etc), information about one of these objects is returned to the caller, and some operation is performed on it to ensure that no other process is triggered by the same object (for details of the structure of the information returned, see below).

If none of the objects is in an available state, the information structure returned will contain an error code to indicate this.

### 6.6.16 Wait on Synchronisation Group, with timeout – sys/kn/sgrp/timedwait

If any of the associated synchronisation objects are in the available state (unlocked for a mutex, positive count for a semaphore, etc), information about one of these objects is returned to the caller, and some operation is performed on it to ensure that no other process is triggered by the object (see below).

If none of the objects is in an available state, the calling process is added to a priority-ordered list of processes waiting for one of the objects in the synchronisation group to become available. If the specified timeout period expires before one of the synchronisation objects in the group becomes available, the information structure returned will contain an error code to indicate this.

## 6.7 Structure of Synchronisation Group Information records

Each of the "wait" functions above is passed a data structure which it fills with information about the event that has occurred, including identification of the synchronisation object that caused the wait to be terminated. The structure is a block of memory 16 bytes in length, with contents defined as follows:

The first word of the structure identifies the type of object that has caused the wait to be terminated. It also determines the contents of the remainder of the structure, and the action that was taken when the event occurred. Possible values of the first word are

- EINVAL: Error
  The remainder of the structure is undefined
- ETIMEDOUT: Timeout
  The remainder of the structure is undefined
- ENOLCK: No available synchronisation objects
  The remainder of the structure is undefined
- EINTR: An unmasked signal occurred
  The remainder of the structure is undefined

- 0: Mutex
  The second word of the structure contains a pointer to the mutex that was triggered. The mutex is now locked by the calling process.

- 1: Semaphore
  The second word of the structure contains a pointer to the semaphore that was triggered. The semaphore has been "waited on" by the calling process.

- 2: Mailbox
  The second and third words of the structure contain the mailbox ID of the triggering mailbox. A message has been read from the mailbox, and its pointer is in the fourth word of the structure.

- 3: Synchronisation Group

The second word of the structure contains a pointer to the synchronisation group that was triggered. The third word of the structure contains a pointer to the information structure that was passed in when the synchronisation object was associated.

- 4: Event Flag
  The second word of the structure contains a pointer to the event flag which whose pattern was matched. The matching pattern is in the third word of the structure and the matching mode is in the fourth word of the structure.

# 7. Memory Management

The int**e**nt model for memory allocation is object-oriented, and therefore very flexible and configurable. The int**e**nt "memory object" implements an interface which supports allocation, freeing, increasing and decreasing the size of the memory pool available to the object, structural checking and information gathering. There are a number of different classes which implement this interface, each of which provides a different memory allocation algorithm, designed for particular features such as speed, low memory overhead or determinism.

There are standard memory object "mappings" defined by the int**e**nt system. These are simply pointers to the memory object to be used for a particular set of memory operations. For example, there are mappings defined for the following areas:

- default application data memory object
- default stack memory object
- system data memory object
- mail message memory object
- code memory object

These mappings may all refer to separate memory objects, allowing the different areas of the system's memory usage to be directed to specific memory objects. These may be implemented by different classes, and may therefore use different algorithms for allocation and freeing. Alternatively, the mappings may all point to the same memory object, which may be the only object in the system. As these mappings are defined at sysgen-time, this gives the system integrator freedom to modify the system memory layout without even reassembling.

There is no limit on the number of memory objects that may exist within a particular int**e**nt system. These objects may be defined at sysgen-time, or dynamically created and destroyed at run-time. In the former case, the objects are usually named, whereas in the latter case they tend to be unnamed. A memory object can be made visible to the whole system by giving it a name, or by defining one of the system mappings to point to it. Alternatively, the object may be private to a particular application or set of applications.

During the lifetime of an individual object, the amount of memory available for the object's use may be dynamically increased or decreased. Memory from one object may be used for the creation of another object, thus creating a hierarchy of memory objects. The algorithm used for allocation or deallocation of memory is dependent on the class used, and may be specified at sysgen-time, or when the memory object is created.

For many embedded systems, a fixed amount of memory is determined at sysgen-time, and remains a constant for that particular platform. On others, it may be desirable to operate at boot time as though only a small, fixed amount of memory were available, and later to perform a memory scan to find the real amount of memory present. Any extra memory then found can be added to one of the memory objects in the system. On yet other systems, a host OS may provide memory services. In these cases, it may be possible to allocate extra memory for the int**e**nt system at any time using the host OS's memory services.

Like other int**e**nt objects, memory objects are initialised through the use of a constructor. A constructor is a function belonging to a specific class, and used to initialise and name objects of that class. The constructor used dictates the type of memory object being constructed.

The following memory allocator classes are available:

| Class Name | Functionality |
|---|---|
| *sys/kn/mem/debug* | Debugging memory object which checks for corrupt free blocks, writes off the end and start of allocated blocks, etc. By default, debug extends the gwc allocator, but can be customised to inherit from any other memory allocator.<br><br>Since this memory allocator reduces performance, and increases memory usage, it should be used only for debugging purposes. |
| *sys/kn/mem/gwc* | 'Good Worst Case' allocator. Provides deterministic best-fit allocation and constant-time freeing. This is the default allocator for systems which do not rely on the PII for allocation and is particularly recommended for heavy loads or where fragmentation is a concern. |
| *sys/kn/mem/pii* | Calls down to PII (underlying OS) for each allocate and free. Performance and determinism entirely dependent on underlying OS. |
| *sys/kn/mem/bp* | This 'bomb proof' allocator is based on the GWC allocator but is modified to assist detection of memory-related errors. It is slower than GWC and has a higher per-block memory overhead. This allocator is also capable of repairing some types of memory corruption in order to prevent crashes. This allocator is only intended for development use. |

## 7.1 Dynamic Memory Allocation

### 7.1.1 Allocate memory from specified type of memory object - *sys/kn/mem/alloc<type>*

The flexibility of the intent system allows for a variety of different underlying memory configurations.

An intent system may contain several memory objects, dedicated to different purposes. For example, there may be one area of memory for process stacks, another for system data, and yet another for general application data. As another example, a system may be designed so that a separate memory object exists for each application, using memory taken from the system default memory object. Alternatively, a system can be configured so that it contains only one memory object.

Such details are invisible to the typical intent application, which is simply offered a range of kernel tools that allow it to allocate memory for particular purposes. Provided the correct tool is used to allocate particular types of memory, the calling code will work irrespective of the memory configuration of the system.

These tools are listed below:

- *sys/kn/mem/allocstk*
  Used by the intent system to allocate memory for process stacks.
- *sys/kn/mem/allocdata*
  Used by applications for allocating data which does not need to persist after the allocation process terminates.
- *sys/kn/mem/allocsys*
  Used to allocate data memory for the intent system itself.
- *sys/kn/mem/alloccode*
  Used to allocate memory to store dynamically loaded tools.
- *sys/kn/mem/allocmail*
  Used to allocate memory to be sent as mail messages. Having a separate object for this allows for the possibility of mapping memory between memory spaces on platforms which support this.
- *sys/kn/mem/allocdef*

Used by libraries to allocate data memory which may need to be shared with other processes, and which must exist after the allocating process terminates.

All of these tools share the same interface, as documented above.

It is important to allocate from the correct memory object for various reasons, the most important being the occasional need for an object to persist after the termination of the process which allocated it.

For example, it is possible for each process in a system to have a separate memory object for its default data allocator. The function *sys/kn/mem/allocdata* is suitable for allocation of a process's data, which does not need to persist after the process's termination. However, a library function called by one of the processes might need to allocate memory which must 'outlive' the allocating process. In such a case *sys/kn/mem/allocdata* would be unsuitable, whereas the *sys/kn/mem/allocdef* function is designed for just this sort of allocation.

The tools described above all call down to the appropriate memory object through an intermediate level interface, *sys/kn/mem/alloc*. This function records the memory object from which each block is allocated. Thus, when it is desirable to free blocks of memory back to the memory object from which they were allocated, it is not necessary for the system to contain a range of separate routines for freeing memory, each corresponding to a different function in the list above. Instead, there is a single tool *sys/kn/mem/free* which automatically frees the memory to the correct memory object, using the record made by *sys/kn/mem/alloc*.

Memory blocks returned by these tools are always 8-aligned and always a multiple of 8 bytes in size.

### 7.1.2 Allocates memory with a specified alignment - sys/kn/mem/allocaligned

#### Inputs

- i<bytes>: number of bytes required
- i<alignment>: alignment required (must be a power of 2)

#### Outputs

- p<mem>: pointer to allocated memory (NULL if failed)
- i<actualbytes>: size of allocated memory

This function allocates a memory block. *sys/kn/mem/allocaligned* takes two parameters, one which dictates the size of the block, and one which specifies the alignment. For example, if the ‘alignment’ parameter is 32, the returned memory block will be 32-byte aligned. *sys/kn/mem/allocaligned* allocates this block from the system memory allocator, in the same manner as *sys/kn/mem/allocsys*.

The function returns two parameters, one of which gives the size of the allocated memory, and one of which is either a pointer to the allocated memory, or is null to indicate a failure. It should be noted that, since this tool returns two values, if it is called from C the multiple return registers facility of the Elate C compiler will need to be used.

### 7.1.3 Allocate memory from specified memory object - sys/kn/mem/alloc

This function is used for allocating memory from a specified memory object. It can be used, for example, by a process which is using a dynamically created memory object.

When an application-level memory routine is used to allocate memory from a memory object, this tool records the size of the allocated block, and the memory object from which the block was allocated. These details, therefore, do not need to be specified when the block is freed back to the appropriate object.

Due to this, the API for freeing a block which was allocated using this function is very simple (see *sys/kn/mem/free*).

If the specified memory object returns a failure, the tool *sys/kn/mem/flush* is called, and the allocation attempted again. If the memory object still returns a failure, this is returned to the caller.

Memory blocks returned by this tool are always 8-aligned and always a multiple of 8 bytes in size.

### 7.1.4 Free memory allocated from corresponding allocation tool - sys/kn/mem/free

This tool frees the specified block, which must have been allocated using *sys/kn/mem/alloc*, *sys/kn/mem/realloc* or one of the *sys/kn/mem/alloc** tools. Using the record made by *sys/kn/mem/alloc*, it automatically frees the block to the correct memory object, releasing the same number of bytes as was originally allocated.

If a NULL pointer is passed in, the function returns immediately.

### 7.1.5 Re-allocate memory - sys/kn/mem/realloc

This tool returns a memory block which is at least as large as the specified size, containing the same data as the block passed in, up to the size of the original block.

If the input block is large enough to satisfy the request, the returned block will be identical to the input block. If the requested block of memory is larger than the input block, the returned block will contain the input block data plus enough memory to meet the request.

If the returned block is not the same as the input block, the input block will be freed automatically before this call returns. It should therefore never be used by the calling process again.

If the input block is not large enough to satisfy the request and memory cannot be allocated to meet the request, NULL is returned as the pointer to the new block. In this case, the input block is still valid and may still be used by the caller (and indeed must still be freed by the caller when it is no longer required).

### 7.1.6 Check structure of all system memory objects - sys/kn/mem/check

This tool calls the check method of all memory objects known by the int**e**nt system. This includes any named memory objects in the sysgen instruction file that have manager tools defined. It also includes any memory objects for which system mappings are defined.

The check method checks the consistency of the memory object; further details can be found below in the section on “Memory Object Methods.”

### 7.1.7 Get pointer to named memory object - sys/kn/mem/lookup

Memory objects defined at sysgen-time are named. This function returns a pointer to the memory object with the specified name.

Having received this pointer, the caller may then directly call the memory object using any of the methods documented in the section "Memory Object Methods," or it may use the *sys/kn/mem/alloc* tool to allocate memory from it. Memory allocated by directly calling the methods of the memory object must be freed in the same way, and can neither be freed using *sys/kn/mem/free* nor reallocated using *sys/kn/mem/realloc*.

### 7.1.8 Return size of block of memory - sys/kn/mem/size

Given a pointer to a block of memory allocated by *sys/kn/mem/alloc* or any of the *sys/kn/mem/alloc<type>* functions, *sys/kn/mem/size* returns the size in bytes of usable memory in the block. The size returned will be the same as that returned by the original call to the allocation function.

It should be noted that this function can only be called with the exact pointer returned by the allocation function. Other pointers, including pointers to other locations within the same block, will cause the function to return undefined values.

### 7.1.9 Returns the memory object associated with a block - sys/kn/mem/getobj

Given a pointer to a block of memory returned by *sys/kn/mem/alloc* or any of the *sys/kn/mem/alloc<type>* tools, this function returns a pointer to the memory object from which the block was allocated.

It should be noted that this tool may only be called with the exact pointer returned from the allocation function. Other pointers, including pointers to locations within the same block, will cause this function to return undefined values.

## 7.2 Virtual Memory Services

int**e**nt is generally used in embedded applications, for which virtual memory capabilities are often undesirable. As a consequence, the majority of platforms upon which int**e**nt is run do not provide a virtual memory system. However, since such facilities are available in platforms such as Linux, DOS DPMI and Win32, the int**e**nt kernel contains a few functions which may be used to control the virtual memory system.

The only functions required by the programmer are those for locking down blocks of memory, so that they are not paged out during periods of high memory usage. Blocks of memory that are paged out to disk may be re-used, and their contents altered. In some cases this is undesirable.

This particularly applies to the writing of interrupt handlers, since many platforms will not support page loading during an interrupt service routine. Therefore interrupt handlers must only use memory that has already been locked. Neither code nor data memory used by the interrupt handler may be allowed to be 'swapped out' by the virtual memory system. This avoids the following situation:

Like a virtual memory paging, an interrupt induces the processor to save the state of the halted operation, and jump to another part of the code. If the appropriate handler for an interrupt is 'swapped out' when the interrupt occurs, the system needs at once to save the state of the interrupted operation, and to retrieve the interrupt handler using the virtual memory paging system. In such circumstances, the state of the system may become confused.

### 7.2.1 Lock Memory - sys/kn/mem/lock

This function locks the specified region of memory into physical memory, so that it is no longer eligible for paging.

This function should be used with care in any system that uses virtual memory since, if too much memory is locked, the system may 'thrash.' Thrashing occurs where the operation of a system is seriously slowed through a shortage of free memory resources. To use virtual memory, a system must save the contents of memory to disk in order to free memory space, retrieving it from disk again when it is needed. A system using virtual memory is said to thrash when there is insufficient unlocked memory to provide a reasonable sized working set for normal memory operations, a situation that forces repeated saves to and loads from disk.

The parameters passed to *sys/kn/mem/lock* specify the address of the start of the region to be locked, and the length of the region in bytes.

*sys/kn/mem/lock* may be called repeatedly to lock the same memory area, without the region needing to be unlocked between these calls. These locks are nested, and a count of multiply locked blocks is maintained. A block which is locked multiple times must be unlocked the same number of times in order to be eligible for paging.

### 7.2.2 Unlock Memory - sys/kn/mem/unlock

This function unlocks the specified region of memory. If the region, or any page within it, becomes completely unlocked, it is eligible to be paged by the virtual memory system.

The parameters passed in to the function specify the address of the start of the region to unlock, and the length of the region in bytes.

The *sys/kn/mem/unlock* function must only be called to unlock memory regions which were previously locked using *sys/kn/mem/lock*. If the specified region of memory is not locked, EINVAL is returned.

## 7.3 Memory Object Methods

### 7.3.1 Constructor tool - <class name>/_new

The constructor is normally the only tool which is called by name to act upon a memory object. All other tools that operate upon memory objects are called using the *ncall* mechanism, isolating the programmer from the details of the memory object type.

The constructor tool takes a pointer to the address at the start of the region from which memory may be allocated. The tool also takes a parameter indicating the length of the specified memory area. In some cases it may be possible for the allocator to acquire its memory pool from another source, so that these parameters are not needed. In the case of a hosted environment, for example, the memory block may be supplied by the host OS.

Most object constructors take no parameters, and return a pointer to an *ncall*able object which they have allocated. This is obviously impossible for a memory object constructor, as the only memory available for allocation may exist within the pool of memory which is being set aside for the object.

The parameters to the memory object, therefore, are not passed straight to the object's *_init* method as would usually be the case. Instead the parameters are passed into the object's *_new* tool, or constructor tool. This tool takes on the task of allocating memory for the object. The *_init* method still exists, and should perform most of the object initialisation. The *_new* tool should simply set up the object so that the *_init* method can be called.

The following methods are supported by memory objects:

### 7.3.2 Initialise memory object - _init

This method initialises the specified memory object in a class-specific manner.

### 7.3.3 Allocate memory from memory object - alloc (xmethod)

This method attempts to allocate the specified number of bytes from the relevant memory object.

If successful, the function returns a pointer to the allocated memory. It returns 0 if the allocation failed.

No memory object will successfully allocate less than 16 bytes. If less than 16 bytes is requested, the request size is rounded up to at least 16 bytes. Memory objects will only return memory blocks whose size is a multiple of 8 and the start of the blocks should be aligned to an 8-byte boundary.

### 7.3.4 Free memory to memory object - free (xmethod)

This method releases the specified memory block, making it available for re-allocation. If the block passed to this method was not originally allocated from this memory object, the results are undefined.

### 7.3.5 Add memory block to memory object - addblock

This method adds a block of memory to the pool of the specified memory object. This allows the object to allocate from this block.

### 7.3.6 Return statistics about memory usage - info

This method returns information about the memory usage of the memory object. It returns the total amount of memory in the pool, the amount of available memory, the size of the largest available block, and the number of available memory blocks.

### 7.3.7 Check memory object structure - check

This method is used for debugging, and checks the consistency of the memory object's structures. The method may be called by an application when memory corruption is suspected. If an error in the object's structures is found, the memory object should call *sys/cii/breakpt* after printing a diagnostic message. This CII function creates a breakpoint within the code, and thus execution is halted if a fault is found by this method.

In this way, buffer overruns, invalid pointer uses, etc, may be isolated by inserting numerous calls to this method within their code.

### 7.3.8 Return size of largest available memory block - largest

This method returns a lower bound on the size (in bytes) of the largest block of memory currently available for allocation by the memory object.

If only the ‘largest block’ memory statistic is required, the largest method removes the need to call the much more heavyweight *info* method.

## 7.4 Memory Flushing

This facility permits applications, device drivers and other code within the intent system to be called and flushed in the event of a memory shortage. Thus, programs are able to implement their own caches in the fashion suited to each. This is achieved by the addition of a "memory flush node" to the kernel's memory flush list, which is processed when resources are low.

The memory flush list is comprised of a series of callback nodes. A callback node is a data structure designed to notify a callback handler of the occurrence of a specific event. In this case, the event in question occurs when the system runs low on memory. These nodes can be found described in greater detail in the account of 'Memory Flush List Nodes' in the section on data structures later in this document. (It should be noted that here the 'callback' is not of the format used by the *sys/kn/callback* functions.)

Each node contains the address of a callback handler function to be called when the system is low on memory. This function is called using a pointer to its memory flush list (MFL_) node, and it returns nothing.

When the memory runs low, the kernel calls the *sys/kn/mem/flush* function to process the nodes in the flush list. The callback nodes then notify their handlers, and the areas of memory that they represent are freed.

### 7.4.1 Add callback routine to memory flush list - sys/kn/mem/addflush

This function allows a callback node to be added to the kernel's flush list. This list is processed by the *sys/kn/mem/flush* function when the kernel finds that it has run out of memory.

It is the caller's responsibility to set not only the MFL_HANDLER field, which must point to the flush callback routine, but also two other fields: MFL_MEMOBJ and MFL_PRIORITY. If the node handles flushes for a specific kernel memory object, then MFL_MEMOBJ must be a pointer to this object. Alternatively, if this handler is called by all flushes, then MFL_MEMOBJ just be NULL. MFL_PRIORITY is an integer which affects the order in which handlers are called when attempting to free up memory. If enough memory is freed by earlier handlers, then lower priority handlers may never be called. The value of MFL_PRIORITY should normally only be set to one of: MFP_PREFER, MFP_OFTEN, MFP_NORMAL, MFP_SELDOM OR MFP_EMERGENCY. If in doubt, MFP_NORMAL should be used.

The flush callback routine must not call any tool which might attempt to allocate memory. This will generally mean that no kernel functions should be called other than *sys/kn/mem/free*. In addition, no function should be called using VIRTUAL and VIRTUAL+FIXUP, since these mechanisms both implicitly allocate memory.

If the routine must access data structures protected by a mutex, it should be prepared for the fact that the calling process may already own the mutex, in which case EDEADLK will be returned. In such circumstances, the callback handler should generally abort without accessing the data structures.

The callback routine may be called from within any process which attempts to allocate memory, and not necessarily from within the process which originally placed it on the flush list. It returns a lower bound on the amount of memory freed by flushing, which should be the exact amount if it is known.

### 7.4.2 Remove a callback function from the memory flush list - sys/kn/mem/removeflush

This function removes a callback node from the kernel's flush list. This list is processed by the *sys/kn/mem/flush* function when the kernel finds that it has run out of memory.

### 7.4.3 Process memory flush callback list - sys/kn/mem/flush

This function is called by the kernel when the system is low on available memory. It may also be called by applications at any time, although this is not commonly done.

It calls the callback routines specified in each of the MFL_ nodes on the callback list, allowing them to free memory. Nodes are traversed in decreasing order of MFL_PRIORITY. If a memory object pointer is passed to this tool then only nodes with matching MFL_MEMOBJ (or NULL) will be considered. If the target amount of memory is freed then no further nodes are visited.

# 8. Timer Management

intent provides two types of timers, periodic and monoshot.

A monoshot timer expires after the specified time, and then becomes inactive.

A periodic timer expires after the specified initial interval. Its expiry time is then reset to the amount of time specified as its 'period.' The timer's expiry time is reset to its period after each subsequent expiry, until it is disabled by a call to *sys/kn/timer/unset.*

In either case, at the expiration of the timer, an application specified action is performed. The action can be one of the following:

- Wake a specified process
- Send a signal to a specified process
- Call a specified function - a timer handler

Timers have two main uses. The first is as a means of providing application or device specific timing information. When transmitting packets of data across networks, for example, timers may be used to make sure that data is not lost in transit. Packets can be given deadlines, so that if they have not reached their destination by a certain time, they will be retransmitted.

Timers may also aid system processing by, for example, rescheduling processes or providing deadline violation detection.

The same mechanism for timer handling is used by applications as by the system.

A timer is a data structure, and is defined in greater detail in the section on data structures later in this document. When a timer is set, it is linked onto a list of timers. This list is ordered primarily by the priority of the timer owner process, and secondarily by the expiry time of the timer, as shown in Diagram 7.

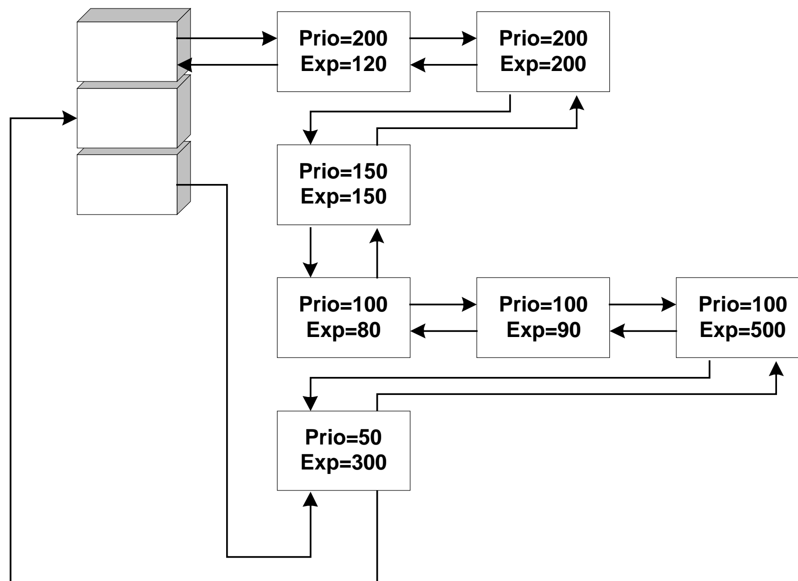


**Diagram 7 - Timer List**

The expiry times shown in Diagram 7 are fictitious, and are used only to show the ordering of timers on the list.

When a dispatch operation occurs, the list is scanned. Each node is processed until a node is reached whose priority is lower than the priority of the highest runnable process. Thus, the only nodes processed are those with priorities greater than or equal to that of the highest priority runnable process. This prevents a high priority task being stopped while a timer handler executes for a lower priority process (a case of priority inversion).

This has no effect on the scheduling order of the system or the performance of the lower priority process. Even if, technically, the effects of a low priority process timeout should occur during a period when a higher process is running, these effects only become relevant when it is the turn of the lower priority process to run.

When the *sys/kn/timer/set* is called to set up the timer, the priority of the timer handler is set to match that of the owner process. Subsequent changes to the priority levels of the owner process do not affect the priority at which the timer handler executes. If the timer handler for a process needs to increase the priority of the process by calling *sys/kn/proc/setparams* or some similar function, problems can be avoided by making the relevant alteration to the priority of the process before the timer is set up, or by using the *sys/kn/timer/dset function*, which allows the caller to specify the priority of the timer.

If an application timer handler function is specified by a timer, the value specified in the TT_PARAM field of the timer structure is passed in to the timer handler. No other parameters are given, and no return parameters are expected.

Timer handlers can legally call only a restricted set of functions. The handlers do not execute "within" the context of the process which set them up, and are effectively software interrupt handlers. Timer handlers will in fact execute with the GP register set to NULL and with interrupts off, in the same fashion as interrupt handlers. A timer handler should return in the same way as a normal function.

See the account of interrupt handlers, in the documentation on the Platform Isolation Interface, for information about the system functions and macros available for use by interrupt/timer handlers.

### 8.1.1 Set up a timer, using the priority of the calling process - sys/kn/timer/set

This function sets up a timer with the same priority as that of the calling process, using parameters given in the specified timer structure. When the structure is passed to this function, all fields but those containing the link node and the priority should already be complete. The structure should not be modified, re-used or freed by application code until the timer is unset, or until the timer permanently expires (this last may only occur in the case of a monoshot timer).

The timer set up in this fashion may be either monoshot or periodic, and may specify relative or absolute times. All times are specified in nanoseconds, though the system granularity may not be this fine.

This function cannot be called from within a timer handler. To do this, one should use *sys/kn/timer/dset*.

### 8.1.2 Set up a timer, using the priority specified in the timer data structure - sys/kn/timer/dset

This function sets a timer, using the parameters given in the specified timer structure. Unlike the *sys/kn/timer/set*, this function does not give the timer the priority of the calling process. Instead, the timer priority is specified in the structure.

When the structure is passed to this function, all fields but the link node should already be complete. The structure should not be modified, re-used or freed by application code until the timer is unset, or until the timer permanently expires (this last may only occur in the case of a monoshot timer).

The timer set up in this fashion may be either monoshot or periodic, and may specify relative or absolute times. All times are specified in nanoseconds, though the system granularity may not be this fine. This function can be called from within a timer handler.

### 8.1.3 Unset timer - sys/kn/timer/unset

This function removes the specified timer from the system timer list. The input pointer must indicate a valid timer data structure. If it is not, the results are undefined. It is not necessary for the application to ensure that the timer is running at the time of the call, however the application must ensure that the structure is currently in use as a timer.

### 8.1.4 Unsets periodic timer from within its own handler - sys/kn/timer/handler_unset

This function removes the specified periodic timer from the system timer list. The function is passed a pointer, which must point to the correct timer data structure, otherwise the results are undefined. No output parameters are returned. The function should only be called from within the handler of the timer which is to be removed.

# 9. Interrupt Handling

int**e**nt interrupt handlers may be written in VP, native code or a high level language. Most are now written in VP. int**e**nt provides support for deterministic handling of interrupts; this is handled exclusively by the Platform Isolation Interface (PII).

The functions devoted to interrupt handler management are listed below.

- Set up an interrupt handler - *sys/pii/setint*
- Remove an interrupt handler - *sys/pii/unsetint*
- Disable all interrupts - *sys/pii/int_off*
- Enable all interrupts - *sys/pii/int_on*
- Disable specified interrupt - *sys/pii/int_dis*
- Enable specified interrupt - *sys/pii/int_en*
- Restore previous interrupt state - *sys/pii/int_restore*
- Set up a scheduling flag from within an interrupt handler - *sys/pii/sched_op*

These are described in greater detail in the documentation devoted to the PII.

## 9.1 Restrictions on Interrupt Handlers

The behaviour of interrupt handler functions is subject to strict restrictions. If these restrictions are not observed, the results may be unpredictable. These restrictions apply to areas such as stack usage and memory usage, the responsibility for which lies with the PII, and to certain parts of the kernel.

- **Memory Usage**
  All memory accessed by an interrupt handler must be locked using *sys/kn/mem/lock*. This applies to both code and data memory.

- **Code Usage**
  The set of system functions which may be called from an interrupt handler is very limited. User functions may be callable, so long as they adhere to the rules set down for interrupt handlers regarding Stack, Memory and Code Usage. The system functions available are:

  - *sys/kn/int/event/alter*
  - *sys/kn/int/event/alter_fn*
  - *sys/kn/int/evf/set*
  - *sys/kn/int/mbox/send*
  - *sys/kn/int/proc/wake*
  - *sys/kn/int/proc/suspend*
  - *sys/kn/int/proc/terminate*
  - *sys/kn/int/proc/setparams*
  - *sys/kn/int/proc/getparams*
  - *sys/kn/int/proc/resume*
  - *sys/kn/int/reslock/unblock*
  - *sys/kn/int/reslock/unblockall*
  - *sys/kn/int/sem/post*
  - *sys/kn/int/sig/kill*
  - *sys/kn/callback/set*

These tools have the same interfaces as those of the corresponding tools without the "/int" in their names, and documentation can be found in the relevant sections. It should be noted that these tools cannot be called from a normal process context.

Programmers should also be aware that within an interrupt handler the global pointer will always be set to 0. Thus if the tools called rely on accessing the global pointer this must be done in a different way.

# 10. Exception Handling

The Elate kernel provides facilities which enable a process to catch and handle processor exceptions using Third Level Exception Handlers (TLEHs). The mechanism described here applies to processor exceptions. These are typically exceptions generated by the CPU, although a processor exception can be generated by software. A processor exception is distinct from software exceptions thrown by applications using *lib/throw*.

When an exception occurs, the First Level Exception Handler (FLEH) pushes the exception details onto the stack in the form of an EXC structure, then invokes the kernel's Second Level Exception Handler (SLEH). This SLEH then proceeds to call a number of TLEHs in sequence, until the exception has been successfully processed. Once the exception has been handled by a TLEH, the SLEH ceases to call TLEHs.

The FLEH is part of the Platform Isolation Interface (PII), and the SLEH is part of the kernel. TLEHs are written at an application level, using functionality provided by the kernel.

TLEHs may be divided into System and Process TLEHs, or into Debugger and non-Debugger TLEHs. A Debugger TLEH is registered by the debugger. There can be at most one System Debugger TLEH in the system, and at most one Process Debugger TLEH per process. All TLEHs share the same calling convention.

In the event of an exception, the kernel SLEH will first call the System Debugger TLEH, if such exists. The non-Debugger System TLEHs are then called one by one. If a Process Debugger TLEH exists for the relevant process, then this is called next. The non-Debugger Process TLEHs are then called in sequence. If the exception is still unprocessed, the Process Debugger TLEH, if any, is called again. Finally, if a System Debugger TLEH exists within the system, it is called once more.

Each TLEH can handle one or more intent exceptions. An intent process may add further Process TLEHs to its own list of TLEHs. The new handler's place in the list depends upon whether it is 'dynamic' or 'static.'

A dynamic Process TLEH may be considered 'temporary.' It is for the duration of a particular piece of code, such as a tool or library call. If multiple dynamic TLEHs are set they must be unset in the opposite order to the order in which they were set (ie. LIFO). The memory used to store the corresponding list entry is allocated from the stack, so that the kernel can detect and tidy up appropriately if an asynchronous transfer of control (such as a software exception, *longjmp* call or a direct call to *sys/kn/tool/setspngo*) bypasses the normal return path and jumps directly to a higher stack level. This allocation must be done by the code which installs the dynamic exception handler. New handlers of this sort are placed at the head of the list.

Static Process TLEHs are installed for the life of the process, and may be considered permanent. *sys/kn/memallocdef* is used to allocate the memory needed for the TLEH list entry. A new static handler will be positioned in the list after the dynamic entries, but before the other static entries. Thus the latest static entry will be the first static handler to be invoked when an exception occurs.

The list of Process TLEHs, therefore, is structured as shown below. The numbers in Diagram 8 indicate the order in which the handlers of each type were installed.

**Head of list**

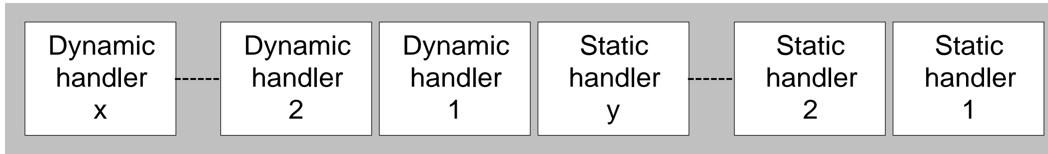


**Diagram 8 – List of Process TLEHs**

A TLEH may modify the fields of the EXC structure. The EXC structure is set up by the FLEH in the case of hardware exceptions, or by the *sys/kn/exc/throw* tool in the case of software exceptions. The structure is then placed on the stack. This takes place before the SLEH is invoked.

A TLEH may, for example, be used to manipulate the EXC_PC field so that, when the FLEH finishes, execution will resume from the new location to which the program counter has been set.

TLEHs operate in an Exception Context under the following restrictions:

- A TLEH may not perform a long-jump, or call any tool that performs a long-jump. This restriction prevents TLEH records remaining on the list after the code for the TLEH has been unloaded. Instead, the Program Counter in the EXC structure is altered to point to the destination of the long-jump. `EXC_SP` should be set to the correct stack position for the destination. The first instruction after the jump should be a 'sync'. The SLEH performs the longjump via a call to *sys/kn/tool/setspngo,* which removes dynamic exception handlers from the stack as they are passed over.

- A TLEH may not remove itself from the list of handlers.

- TLEHs may not call any other tools.

- When a TLEH returns, it must return a code indicating the action the SLEH should take next (see the description of *process_exc_handler*).

## 10.1 Exception Numbers

This table lists the numbers of the intent exceptions for which a TLEH can register.

| intent Exception Number (Hex) and meaning | Signal sent to the process by Kernel SLEH if the exception is not handled |
|---|---|
| 0: Integer divide by zero | SIGKILL |
| 1: Single step | SIGKILL |
| 2: Breakpoint | SIGKILL |
| 3: Invalid OpCode | SIGILL |
| 4: No coprocessor available | SIGFPE |
| 5: Double fault | SIGKILL |
| 6: Stack exception | SIGSTACK |
| 7: General protection error | SIGSEGV |
| 8: Memory fault | SIGKILL |
| 9: Data type misalignment | SIGKILL |
| A: Invalid class method | SIGKILL |
| FFF: General fault (catch-all for remaining | SIGKILL |

| documented exceptions) | |
|---|---|
| 1000 - 1FFF: (Platform dependent exceptions) | SIGKILL |
| FFFF: Unidentifiable exception (catch-all for undocumented exceptions) | SIGKILL |

Exception FFFF is a catch-all for exceptions that are not catered for because they are not documented. All exception numbers not documented in the above ranges are reserved.

## 10.2 The Process/Debugger third level exception handler

The interface for a TLEH is described below.

### 10.2.1 Enables a process to throw a software exception - process_exception_handler

This describes a TLEH which is provided to *sys/kn/exc/set*, *sys/kn/exc/setsys* or *sys/kn/exc/setdbg* in order to catch processor exceptions. The name process_exception_handler is a placeholder for whatever name the programmer chooses.

The first of the parameters passed as an input to this function contains the exception details. The second is the data pointer passed to *sys/kn/exc/set*.

The status code returned must contain one of the following values.

- EXC_HANDLED
  This indicates that the exception has been handled, and that the SLEH should return to the FLEH to resume execution. The TLEH may have modified some of the values in EXC on the stack, in which case these changes must be propagated down to the FLEH. If EXC_PC has been modified, however, the TLEH will return EXC_HANDLED_LONGJUMP rather than EXC_HANDLED.

- EXC_HANDLED_LONGJUMP_VP
  This indicates that the exception has been handled and that the TLEH has modified EXC_PC to effect a long-jump, which will typically be carried out by at the FLEH level. (It may also alter other fields of the EXC structure.) The jump will be carried out indirectly, through the use of *sys/kn/tool/setspngo* by the kernel SLEH. Before the jump, control is passed back to the FLEH, with the EXC_PC field set to the jump routine within the SLEH.

- EXC_HANDLED_LONGJUMP
  This is similar to EXC_HANDLED_LONGJUMP_VP, except that the jump is not handled indirectly. This has certain disadvantages, since in this case *sys/kn/tool/setspngo* will not tidy up the virtual calls and dynamic exception handlers. The advantage of this variant is that the processor's native registers are not corrupted.

- EXC_NOT_HANDLED
  This indicates that the exception has not been handled, and that the SLEH should invoke the next TLEH on the list. If there are no more TLEHs on the list, the SLEH takes the default action. This entails calling the debugger, if one has registered, or calling *sys/kn/sig/kill* with an appropriate signal to the offending intent process. (A list of signals appropriate to each exception is given in the table in the section on "Exception Numbers." No exception details will be available to the signal handler.) This normally results in termination of that intent process.

- EXC_KILL_PROCESS
  This indicates that the process must be killed immediately, without invoking any more TLEHs on the list.

## 10.3 Exception handling functions

The kernel contains a number of functions which allow processes to register and deregister TLEHs.

### 10.3.1 Register a process's exception handler with the kernel - sys/kn/exc/set

This function enables a process to register an exception handler. This handler is added to the list of non-debugger Process TLEHs for that process. If a dynamic handler is being registered, the memory for the TLEH list node should be allocated from the stack.

The function returns two values. The first is a token which can be passed to *sys/kn/exc/unset* when the process no longer wishes to deregister the handler. The second is either an error value, or 0 if the function has completed successfully. It should be noted that since this tool returns two values, if it is called from C it will be necessary to use the multiple return registers facility of the Elate C compiler.

### 10.3.2 Deregister a process's exception handler with the kernel - sys/kn/exc/unset

This function enables a process to deregister an exception handler which was earlier registered by *sys/kn/exc/set*. Thus a specific non-debugger Process TLEH is removed from the list.

This function takes as a parameter a token which should be a value returned from *sys/kn/exc/set*. It returns either an error code, or 0 if the function has completed successfully.

### 10.3.3 Register a system-wide exception handler with the kernel - *sys/kn/exc/setsys*

This function is used to register an non-debugger exception handler, which applies to the whole system as opposed to just a single process.

The function returns two values. The first is a token which can be passed to *sys/kn/exc/unsetsys* when the process no longer wishes to deregister the handler. The second is either an error value, or 0 if the function has completed successfully. It should be noted that, since this tool returns two values, if it is called from C it will be necessary to use the multiple return registers facility of the Elate C compiler.

### 10.3.4 Deregister a system-wide exception handler with the kernel - *sys/kn/exc/unsetsys*

This function is used to deregister a system-wide non-debugger exception handler registered by *sys/kn/exc/setsys*.

### 10.3.5 Register a debugger's exception handler with the kernel - sys/kn/exc/setdbg

This function enables a debugger to register an exception handler. Only one debugger TLEH can be registered. Attempts to register further debugger TLEH's will result in an error code being returned.

The function takes three pointer parameters. The first points to the exception handler. The second provides a pointer to be passed to the exception handler when an exception occurs. The third pointer parameter indicates whether the debugger is to be registered at process-level or system-level.

The function returns two values. The first is a token which can be passed to *sys/kn/exc/unsetdbg* when the process no longer wishes to deregister the handler. The second is either an error value, or 0 if the function has completed successfully. It should be noted that, since this tool returns two values, if

it is called from C it will be necessary to use the multiple return registers facility of the Elate C compiler.

### 10.3.6 Deregister a debugger's exception handler with the kernel - sys/kn/exc/unsetdbg

This function enables a debugger to deregister an exception handler.

The value passed in to the function should be a token returned by *sys/kn/exc/setdbg.*

### 10.3.7 Throw a software exception - sys/kn/exc/throw

This function enables a process to throw a software exception.

# 11. Signals

A signal is a notification of a specific event, sent by the kernel to a process. A range of different signals may be sent, including some with user-defined meanings. Each signal indicates the occurrence of a different event, and is associated with a different signal number. This number may be used to distinguish between the different signals, and is the only information carried by the signal.

A process possesses a different handler for each signal, and can specify actions to be performed when each signal occurs. Each signal has a default action. Details of these can be found listed below.

When the event that causes a signal occurs, that signal is said to be "generated" or “raised.” When an appropriate action is taken as a consequence of a signal reaching the process, the signal is said to be "delivered."

The intent signal mechanism has full functional compatibility with POSIX.1 signals, although the interface to the intent kernel signal function has slightly different return parameters, in order to comply with the intent kernel standard. (In intent, error values are returned in a register to indicate an error. In POSIX, -1 is usually returned, and the global variable error values set, to indicate an error.) The precise POSIX interface can easily be provided as a higher level library layer. POSIX.4 extensions to signals are not supported.

The following signals are defined for intent:

| Symbolic Name | Default Action | Description |
|---|---|---|
| SIGNULL | None | Null signal, can never actually be raised |
| SIGABRT | Terminate | Abnormal termination signal |
| SIGALRM | Terminate | Alarm signal (a POSIX alarm has been triggered) |
| SIGFPE | Terminate | Arithmetic error signal |
| SIGHUP | Terminate | Hangup of controlling terminal |
| SIGILL | Terminate | Invalid instruction |
| SIGINT | Terminate | Interactive attention signal |
| SIGKILL | Terminate | Termination signal (this cannot be caught or ignored) |
| SIGPIPE | Terminate | Write on a pipe with no readers |
| SIGQUIT | Terminate | Interactive termination signal |
| SIGSEGV | Terminate | Invalid memory reference detected |
| SIGTERM | Terminate | Termination signal |
| SIGCHLD | Ignore | Child process terminated |
| SIGUSR1 | Terminate | Application-defined signal 1 |
| SIGUSR2 | Terminate | Application-defined signal 2 |
| SIGUSR3 | Terminate | Application-defined signal 3 |
| SIGTOOL | Terminate | Virtual tool call failed to open destination tool |
| SIGRES | Ignore | Priority inversion detected |
| SIGDLINE | Ignore | Deadline violation detected |
| SIGCONT | Continue process | Job-control continue signal |
| SIGSTOP | Pause process | Job-control stop signal |
| SIGTSTP | Pause process | Job-control interactive stop signal |
| SIGTTIN | Pause process | Job-control signal - background process attempted to read from input |
| SIGTTOU | Pause process | Job-control signal - background process attempted to write to output |

SIGCHLD behaves in a manner different to the other signals. Although this signal is ignored if its handler is set to SIG_DFL (i.e. carry out default behaviour), the child process will be automatically reaped (see *sys/kn/proc/wait*) if the handler is set to SIG_IGN (i.e. ignore signal). This feature provides compatibility with common UNIX systems.

When one of the signals SIGSTOP, SIGTSTP, SIGTTIN or SIGTTOU is generated, any pending SIGCONT signals are discarded. Delivery of these signals causes the target process to stop until a SIGCONT is received. While a process is stopped due to a job-control signal, only SIGCONT and SIGKILL signals will be delivered to the process. When one of the stop signals is delivered to a process, the parent of the target process is sent a SIGCHLD signal.

The generation of a SIGCONT signal causes any pending stop signals to be discarded. It also causes a stopped process to continue even if it is blocked. However in these cases the signal handler, if any, will not be processed until the signal is unblocked.

## 11.1 Generating Signals

A signal is said to be "generated" or "raised" when the event which causes the signal occurs. Events that might cause a signal to be sent to a process can be generally categorised as follows:

- Hardware exceptions
  These include illegal instruction exceptions, invalid memory reference exceptions, arithmetic exceptions, etc.
- Abnormal software conditions
  These include writing to a pipe with no readers, termination of a child process, attempts to virtually call tools which do not exist, etc.
- User-initiated events
  These include the user pressing control-C or control-Y at the keyboard (currently not supported), running the kill program, etc.
- Program-initiated events
  These include the expiration of an alarm, inter-process communication using one of the SIGUSRn signals, etc.

## 11.2 Signal Actions

Each process has a set action with which to respond to each signal defined by the system. An application may individually set the action for each signal supported by the system. There are three possible actions for a signal.

- **SIG_DFL**
  The process may perform the default action for this signal. These actions are described in the table above.
- **SIG_IGN**
  The process may ignore the signal. If this action is taken, the process is unaffected by the delivery of the signal. This action cannot be specified as a response to the SIGKILL signal. If it is set as a response to a SIGSEGV, SIGILL or SIGFPE which was not generated by a call to *sys/kn/sig/kill*, the behaviour of the process is undefined. If the termination of a child process causes the generation of SIGCHLD, and this signal is subsequently ignored, the child process shall be automatically deleted.
- **Pointer to signal handling function**
  On delivery of the signal, the receiving process may execute the user-defined signal handler indicated by a pointer. After execution of the signal handler, the process continues from the point at which it was interrupted. The signal handler is a function which takes a single integer parameter specifying the signal number, and which returns nothing.

A signal handling function may not be set as the response to a SIGKILL signal. If a process returns normally from a signal handling function for a SIGSEGV, SIGILL or SIGFPE which was not generated by a call to *sys/kn/sig/kill*, the ensuing behaviour of a process is undefined.

Instead of returning normally, a signal handling function may use the *longjmp* function to jump to a state recorded earlier using the *setjmp* function.

## 11.3 Delivering a Signal to a Process

A signal is said to be "delivered" to a process when the process's set response to that particular signal is enacted. It is at this point, rather than at the time of generation, that the action to be taken in response to a signal is determined. If the set action is changed during the period between generation and delivery, the revised action will be taken when the signal is delivered.

During the period between a signal's generation and its delivery, it is said to be "pending." Normally this period cannot be detected by the application, as the signal is usually delivered as soon as the process becomes runnable. A signal may be suspended in the pending state for an indefinite period, however, if the signal mask of the receiving process is adjusted. A signal mask exists for each process, and defines the set of signals that are currently blocked, and cannot be delivered to the process. The SIGKILL signal, however, cannot be blocked in this manner.

If a signal is blocked when it is generated, it remains pending until it is unblocked. If the action for a signal is set to ignore, the signal remains pending until it is unblocked and can be delivered, at which point it is ignored. This behaviour is unspecified by POSIX.

A signal will not be generated again for a process that already has that same signal pending. This behaviour is unspecified by POSIX.

When numerous signals are pending for a process, the signal with the lowest number is placed first in the queue. This is unspecified by POSIX.

## 11.4 Disabling Signal Delivery

Many POSIX compliant systems use the concept of a process existing in a "system-state" or a "user-state." Signals are usually only delivered while a process is executing in the "user-state." This provides protection from signal delivery while sensitive operations such as I/O or manipulation of system data structures are being performed.

Although int**e**nt uses no concept of a "system-state" or "user-state," it provides the same sort of protection through a mechanism that makes it possible to "switch off" signal delivery to a process. This does not affect its signal mask, or any pending signals. Most critical regions in the system are protected by mutexes, which provide the MTX_SIGMASK flag. If this flag is set, signals will be switched off automatically by the mutex code when a process becomes the owner of the mutex. Signals are re-enabled when the process unlocks the mutex.

This removes the danger that, during the execution of critical sections of code, the death of a process due to an uncaught signal may compromise the consistency of the data structures, or that the process may longjmp out of a signal handler to a code outside the critical section. If the MTX_SIGMASK flag is used, the critical section is guaranteed to complete before any signal can be delivered which might have these effects.

In most cases signal masks will be sufficient, and applications will not need to use this functionality. It is documented here mainly for the benefit of the authors of device-drivers, and other pieces of code which would conventionally exist in the "system-state."

## 11.5 Signal Functions

### 11.5.1 Send a signal to a process- sys/kn/sig/kill

This function generates the specified signal for the relevant target process. The signal number specified should be one of the signals listed in the table above.

If the signal specified is SIGNULL (zero), error checking is being performed to make sure that it would be possible for the calling process to send a signal to the target process, but no signal is sent. The null signal can be used to check the validity of the specified process ID.

If the specified process ID refers to the calling process, and if the specified signal is not blocked for the calling process, the signal is handled before *sys/kn/sig/kill* returns.

### 11.5.2 Examine or change signal action - sys/kn/sig/action

This function allows the calling process to examine and/or modify its signal handling actions. The function takes three parameters, an integer and two pointer parameters. The integer argument specifies the signal to which the response should be investigated or altered.

If the first pointer is non-NULL, it points to a sigaction data structure specifying the new action to be set for the specified signal. Otherwise, the action for the specified signal is not modified by this call.

If the second pointer is non-NULL, the action associated with the specified signal at the time of the call is stored into the structure to which it points.

The sigaction data structure is defined as follows:

| Name | Description |
|---|---|
| SA_HANDLER | SIG_DFL, SIG_IGN or a pointer to a function |
| SA_MASK | Additional set of signals to be blocked during execution of signal handler function |
| SA_FLAGS | Flags affecting behaviour of signal |

The valid flags in the SA_FLAGS field of the structure are:

- SA_NOMASK
  This indicates that the signal is not blocked by the mask, and does not prevent the signal from being delivered while within its own handler.
- SA_ONESHOT
  This restores the signal action to SIG_DFL after the signal handler has been called. The default behaviour is for the signal action to remain the same after a signal handler executes.

### 11.5.3 Examine or change blocked signals - sys/kn/sig/procmask

This function allows the calling process to examine and/or modify its signal mask, thus altering the set of signals that are blocked from delivery. Sets of signals are expressed within *sigset_t* structures, which are 32 bit integers in which each signal supported by Elate is represented by one bit. The value of each bit indicates whether the associated signal is a member of the group.

The function takes an integer and two pointers as parameters, which specify the action to be performed, as described below.

The value of the integer parameter specifies the operation to be performed on the current set of blocked signals, as follows:

- SIG_BLOCK
  The new set of blocked signals is the union of the current set and the set specified by the *set* parameter.
- SIG_UNBLOCK
  The new set of blocked signals is the intersection of the current set and the complement of the set specified by the *set* parameter.
- SIG_SETMASK
  The new set of blocked signals is the set specified by the *set* parameter.

If the first pointer is non-NULL, it points to a *sigset_t* structure. The current signal mask is a *sigset_t* structure. The new *sigset_structure* indicated is used to alter the old in a fashion specified by the *how* parameter. If the pointer is NULL, the set of blocked signals is not modified by this call.

If the second pointer is non-NULL, the condition of the signal mask at the time of the call is stored into the *sigset_t* structure to which it points.

If any pending signals become unblocked by the call to *sys/kn/sig/procmask*, these signals shall be delivered before the call returns.

### 11.5.4 Examine pending signals - sys/kn/sig/pending

This function examines the set of signals which are blocked from delivery and are pending for the current process. It stores this set into a specified empty *sigset_t* structure. This function will only detect signals that are blocked, either by the signal mask or through disabling of signals. Unblocked signals will always be delivered before it can be called.

### 11.5.5 Return set of signals which have been raised but not yet taken - sys/kn/sig/raised

This function returns the set of signals which have been raised but have not yet been taken by the calling process. This function will only return a non-zero value if signals have been disabled using the *sys/kn/sig/setflag* function.Otherwise, the signals will have been taken immediately upon delivery.

The set of signals returned by this function includes no signals that are blocked using the process's signal mask. While signals are disabled, the operation of the process may only be affected by signals that have not been blocked in this manner.

### 11.5.6 Wait for a signal - sys/kn/sig/suspend

This function replaces the signal mask of the calling process with the specified signal set. The calling process is suspended until the delivery of a signal causes it either to execute a user-specified signal handling function, or to be terminated.

If the process is terminated, this call never returns. If the action is to call a signal handler, this call returns after the signal handler returns, with the signal mask reset to its value before this call.

Since this function suspends the calling process indefinitely, there is no successful completion value. -1 is always returned.

### 11.5.7 Disable or enable signal handling - sys/kn/sig/setflag

This function allows the caller to disable the int**e**nt signal mechanism for the calling process. This is usually a temporary measure adopted during critical sections of code where the occurrence of

unpredictable actions such as process termination, or *longjmp*ing from a signal handler, could leave system data structures in an inconsistent state, and cause a system crash.

The flag values are:

0: Enable signals
1: Disable signals

Usually, the previous state of the signal flag is restored at the end of a critical section, rather than simply using the value 0. This allows for nesting of these operations, without unpredictable effects.

This function should not be called to enable signals when interrupts are off, as this may cause unpredictable behaviour.

### 11.5.8 Sets handler for specified signal - sys/kn/sig/signal

This function is used to set the handler for a particular signal. The function is passed a pointer to the handler, which must point to one of SIG_DFL, SIG_IGN, or to a user-defined handler.

SIG_DFL causes the default action to be performed, SIG_IGN the signal to be ignored. If a user-defined handler is supplied, then on delivery of the signal the receiving process may execute the handler. After execution of the signal handler, the process continues from the point at which it was interrupted. See the earlier section on “Signal Actions” for more details, and for information on special cases.

It is possible to use the same signal handler for several signals.

## 11.6 Signal set functions

These functions manipulate sets of signals. A set of signals is expressed within a *sigset_t* structure, a block of memory of size SIG_SIZE. An empty set contains no signals, a full set contains all signals, and a partial set contains one or more, but not all.

### 11.6.1 Creates an empty set - sys/kn/sig/emptyset

The *sys/kn/sig/emptyset* function initialises the specified signal set structure in such manner that no signals supported by the int**e**nt system are members of the set.

### 11.6.2 Creates a full set - sys/kn/sig/fillset

The *sys/kn/sig/fillset* function initialises the signal set indicated by its argument set in such manner that all signals supported by the int**e**nt system are members of the set.

Applications should call either *sys/kn/sig/emptyset* or *sys/kn/sig/fillset* at least once for each data structure of type *sigset_t* prior to its use as a parameter to any other signal function. If this is not done, the results are undefined.

### 11.6.3 Adds a signal to a set - sys/kn/sig/addset

The *sys/kn/sig/addset* function adds the signal with the specified number to the specified signal set.

### 11.6.4 Delete a signal from the set - sys/kn/sig/delset

The *sys/kn/sig/delset* function deletes the signal with the specified number from the specified signal set.

### 11.6.5 Tests set to see if signal is a member - sys/kn/sig/ismember

The *sys/kn/sig/ismember* function tests whether the indicated signal is a member of the specified set.

### 11.6.6 Modifies the set of pending signals - sys/kn/sig/setpending

This function allows the caller to modify the set of signals which are currently pending for the calling process. The new set is specified by the *sigset_t* passed in as the first parameter. If the second parameter is non-NULL, it should point to a *sigset_t* structure, which is filled with the previous set of pending signals.

This function takes no account of whether the specified signals are blocked. It is possible to set the pending mask to include signals which are blocked or non-blocked, and the returned signal mask will contain both blocked and non-blocked signals. If the set is changed to include a signal which is not blocked, the signal(s) will be taken immediately, unless signal handling has been disabled using *sys/kn/sig/setflag*.

This facility is most commonly used by device drivers that need to remove a signal temporarily from the pending signal set, in order to complete their operations. This allows the device drivers to block on calls to *sys/kn/proc/sleep* without returning immediately due to having signals pending. The signals are usually added back to the pending signal set before the device driver re-enables signals.

### 11.6.7 Modifies the set of pending signals - *sys/kn/sig/orpending*

This function allows the caller to modify the set of signals which are currently pending for the calling process. The new set is set to the union of a specified *sigset_t* structure, and the current set of pending signals. If the second parameter is non-NULL, it is assumed to point to a *sigset_t* structure and filled with the previous set of pending signals.

This function takes no account of whether the specified signals are blocked. It is possible to set the pending mask to include signals which are blocked or non-blocked, and the returned signal mask will contain both blocked and non-blocked signals. If the set of pending signals is changed to include a signal which is not blocked, the signal will be taken immediately (unless signal handling is disabled using *sys/kn/sig/setflag*).

This facility is most commonly used by device drivers to restore a pending signal which has previously been removed.

# 12. Event Tools

Many programs commonly require that some event takes place as part of a process, for which other processes need to wait. For example, processes might need to wait for a shared resource to be available, or for the results of a computation.

Sometimes many processes may need to wait simultaneously for a single event. When the event occurs, one, many or all of the waiting processes may be able to unblock. Sometimes a process may wish to wait for a certain pattern of events to have occurred. The events in question may be permanent, in the sense that they are associated with some sort of state change, or they may be temporary. All of these possibilities and many more are covered by the kernel event tools.

The event tools use a system of 'event trackers,' upon which processes can wait for an event to occur or for a certain state to be reached. Each event tracker holds 32 bits of internal state the semantics of which can vary. For example, it might be used as a 32-bit integer counter, or as a set of 32 boolean flags or even as the "p2i" value of a pointer, which could point to a block of memory whose contents are to be used as the state.

A process may alter the state of an event tracker. If any processes which are waiting are satisfied as a result, they consequently unblock. It should be noted that only calls to the alteration functions *sys/kn/event/alter* and *sys/kn/event/alter_fn* can unblock a waiting process. Other changes to the state of the event tracker will not result in unblocking.

Waiting processes which are unblocked by an alteration to the state of the event tracker may adjust the state of the tracker themselves as part of exiting the wait. For example, a process which had been waiting on a semaphore would decrement the semaphore's count when unblocked.

The event tools do not have a built in priority inversion prevention mechanism, such as that provided by mutexes.

The event tools switch off interrupts as a mechanism for preventing simultaneous modification of the event tracker state by two or more processes. When this is done, interrupts are restored afterwards to their state on entry to the tool.

### 12.1.1 Initialises an event tracker - sys/kn/event/init

This tool initialises an event tracker structure.

The tool is passed a pointer to the event tracker structure to be initialised. This should be a block of memory ETOOLS_SIZE bytes in length or sizeof(ELATE_ETOOLS) if using C).

Another pointer points to the default ‘state change function,’ which is designed to transform the event tracker state. The default state change function for a semaphore, for example, would simply increment the input state. There is no restriction on the use of the data value used by a state change function, which is simply an integer chosen by the caller. It should be noted that a state change function may not switch interrupts on at any point since it may be called within an interrupt handler routine.

Another pointer indicates a different state change function, the ‘reset’ function. This is designed to be called after state alteration. The ‘reset’ function is called by the *sys/kn/event/alter* tool after all appropriate unblocking of waiting processes has taken place. For example, if a particular event were only relevant at the time of the call then the ‘reset’ function might ignore its input state and return 0.

A fourth pointer points to the default 'state checking function' for this event tracker. A state checking function is used to decide whether to unblock a process waiting on an event handler, based on the state of the handler and a value passed in. The checking function should return 0 if the check passes, indicating that unblocking should take place. Otherwise it should return an error code which indicates why it cannot unblock, typically EBUSY to signify a busy resource. The state checking function returns one other parameter, which specifies the new state of the event tracker. Like a state change function, a state checking function may not switch interrupts on at any point since it may be called within an interrupt handler routine.

Here, the state checking function is called by the various wait tools (*sys/kn/event/wait*, *sys/kn/event/trywait*, *sys/kn/event/timedwait,* *sys/kn/event/wait_fn*, *sys/kn/event/trywait_fn*, *sys/kn/event/timedwait_fn*) to test whether it is possible to unblock. For example, the checking function for a semaphore makes no use of its second parameter and returns 0 if its first parameter is greater than 0 or EBUSY. If it returns 0, it also decrements the semaphore count. Since the state checking function returns two parameters, it is therefore it is necessary to use the multiple return registers facility if programming in C.

It should be noted that, since these the default state change function, the reset function and the default state checking function are called with interrupts off, they must perform only brief computations to avoid affecting system performance.

The tool is also passed a flags parameter, which controls the order in which blocked processes are unblocked. Currently only the least significant bit is used:

- Bit 0 clear - ETOOLS_FIFO : Specifies that when processes block waiting for an event, they should be unblocked in a first-in-first-out order.
- Bit 0 set - ETOOLS_PRIO : Specifies that when processes block waiting for the event, they should be unblocked in a highest priority first order. If multiple processes of the same priority are blocked on the same Event tracker, they are unblocked in a first-in-first-out order.

The last parameter passed to the tool specifies the initial state of the event tracker.

### 12.1.2 Destroys an event tracker - sys/kn/event/destroy

This function is used to destroy the specified event tracker. Only an event tracker which was initialised using *sys/kn/event/init* may be destroyed using this function.

The effect of subsequent use of the specified event tracker is undefined, unless it has been re-initialised by another call to *sys/kn/event/init*.

Destroying an event tracker that has processes waiting on it has the effect of unblocking all of those processes and returning the error code EINVAL to them.

### 12.1.3 Waits on an event tracker (blocking, no timeout) - sys/kn/event/wait

This function checks the state of the specified event tracker. The check is carried out by calling the default state checking function with which the event tracker was initialised. The calling process blocks if the checking function does not return success (ie. does not return 0). Unblocking can only occur via a call to *sys/kn/event/alter* or *sys/kn/event/alter_fn*. Note that it is undefined how often the checking function is called on a blocked wait, so in general the state should not be changed from the input value unless the check has passed. Although it is not forbidden to change it in other cases, this is usually undesirable.

### 12.1.4 Waits on an event tracker (non-blocking) - sys/kn/event/trywait

This tool behaves similarly to *sys/kn/event/wait* except that the checking function is called only once and if the check fails, the tool simply returns an error (typically EBUSY). This tool never blocks.

### 12.1.5 Waits on an Event tracker (blocking, with timeout) - sys/kn/event/timedwait

This tool behaves similarly to *sys/kn/event/wait* except that after a specified period of time has expired, the tool ceases to wait and returns ETIMEDOUT.

### 12.1.6 Alters the state of an event tracker - sys/kn/event/alter

The default state change function associated with the event tracker is called on the event tracker's current state. This may permit the unblocking of one or more waiting processes. Waiting processes are considered one by one in order, since the unblocking of each may change the state further. The order in which the blocked processes are considered is determined by the ETOOLS_FIFO/ETOOLS_PRIO flag passed to *sys/kn/event/init*.

An 'interrupt-safe' version of this tool, *sys/kn/int/event/alter*, is also available. *sys/kn/int/event/alter* can safely be called from inside an interrupt. Further details are available in the later section on “Interrupt Handling.”

### 12.1.7 Gets the value of an event tracker - sys/kn/event/info

This function returns the value of the specified Event tracker's state.

### 12.1.8 Waits on an event tracker in a specified way (blocking, no timeout) - sys/kn/event/wait_fn

This tool behaves identically to *sys/kn/event/wait*, except that it takes an extra parameter which affects the operation. This parameter is a pointer which specifies a state checking function to use in place of that specified at initialisation time. This substitution applies to this call only, and the old function is not permanently replaced.

### 12.1.9 Waits on an event tracker in a specified way (non-blocking) - sys/kn/event/trywait_fn

This tool behaves identically to *sys/kn/event/trywait* except that it takes an extra parameter which affects the operation. This parameter is a pointer which specifies a state checking function to use in place of that specified at initialisation time. This substitution applies to this call only, and the old function is not permanently replaced.

### 12.1.10 Waits on an event tracker in a specified way (blocking, with timeout) - sys/kn/event/timedwait_fn

This tool behaves identically to *sys/kn/event/timedwait* except that it takes an extra parameter which affects the operation. This parameter is a pointer which specifies a state checking function to use in place of that specified at initialisation time. This substitution applies to this call only, and the old function is not permanently replaced.

### 12.1.11 Alters the state of an event tracker in a specified way - sys/kn/event/alter_fn

This tool behaves identically to *sys/kn/event/alter* except that it takes two extra parameters which affect the operation. The first is a pointer which specifies a state checking function to use in place of that specified at initialisation time. This substitution applies to this call only, and the old function is not permanently replaced. The second extra parameter replaces, in a similar fashion, the default state change function called after all checks for unblocking of processes are complete.

An 'interrupt-safe' version of this tool, *sys/kn/int/event/alter_fn*, is also available. *sys/kn/int/event/alter_fn* can safely be called from inside an interrupt. Further details are available in the later section on “Interrupt Handling.”

### 12.1.12 Call a function for each caller waiting on an event tracker - sys/kn/event/enumerate

This tool calls an ‘enumeration function’ upon the event tracker, once for each blocked call waiting on the event tracker. The calls are made in an order determined by the order in which the blocked calls would be considered following a call to *sys/kn/event/alter*.

The enumeration function returns a parameter which specifies the new state of the event tracker, and thus each call to an enumeration function may change the state of the event tracker. The enumeration function can also alter a value parameter, which can be used therefore to count the number of blocked waits.

A typical use of this tool is to correct the state of the event tracker when a timed wait times out and the corresponding blocked call is removed from the list. It may at that point be necessary to assess the remaining blocked calls in order to determine the new state.

# 13. Callback Manipulation Functions

Within intent, a device driver or another piece of code is able to register a function to be called within the context of a specific process.

Device drivers often use callbacks to notify an application of the completion of I/O operations that have been executed in an asynchronous fashion. When an application is writing a quantity of data to a device, it may often be undesirable for the application to wait until this task is complete. If a callback is set up, the application specifies a function to be called when the task is complete. It is then able to proceed with other tasks, until the specified function is called. Thus the application is notified that the task is complete, and that the buffer holding the relevant data may be safely reused.

A callback is defined by a data structure which contains information such as the address of the function to be called back, the process ID of the process in whose context the callback is to be executed, and a data pointer to be passed to the callback handler function.

A process requests that a callback function be executed within its context by "setting a callback." To do this, the process calls the kernel callback manipulation function *sys/kn/callback/set* (see below). The data structure that defines the callback function is placed on the relevant process's callback list so that, the next time the process is idle, a call to the specified callback function will be triggered.

Whereas signals are executed asynchronously with program execution, and therefore may be delivered at any time that they are not blocked on a mutex or a signal mask, callbacks only execute under specific circumstances.

Callbacks may be called when a process halts its own execution by calling *sys/kn/proc/sleep* or *sys/kn/proc/deschedule*, unless callback processing has been disabled using *sys/kn/callback/setflag*. (It is desirable that the callback should execute while the process is idle, so that it cannot interfere with the deterministic performance of the process's other activities.) However, it is also possible for the programmer to make a direct call to *sys/kn/callback/process* and thus process any outstanding callbacks without surrendering CPU time. Note that if *sys/kn/callback/process* is called directly, callbacks are processed regardless of whether callbacks are nominally enabled or disabled.

If a process with pending callbacks exits, the callbacks will never be taken. It is safe to free any associated resources, for example memory used for the callback structure, in this case.

Just as signals may be 'switched off' to protect sensitive operations from signal delivery, the execution of callbacks can be disabled through use of the kernel function *sys/kn/callback/setflag*. The processing of callbacks during the execution of some system functions or device drivers can create a risk of deadlocks or race conditions. If this is the case, callbacks should be disabled throughout the execution of the function or driver.

## 13.1 The Callback Data Structure

Callback functions are defined by callback data structures, which are described in greater detail in the "Data Structure Definitions" section later in this document.

Each structure contains the list node used to link the structure onto the process's callback list, a data pointer to be passed to the callback function to be called, the address of the callback function, and the identifier of the process within the context of which the callback is to be executed.

When a callback is "set" by a process, this structure is placed on the process's callback list.

## 13.2 The Callback Handler Function

A callback function is in effect a handler for a specific event.

A callback handler function takes as parameters a pointer to the associated callback data structure, and the value in the CALLBACK_DATA field of this structure. The function has no return values.

The callback handler function executes in the context of the process which was specified in the CALLBACK_PID field of the callback data structure when *sys/kn/callback/set* was called.

During a callback, the intent system is in a state similar to that during normal execution. There are no restrictions upon which functions may be called while a callback handler is executing.

Any system function or device driver which wishes to prevent callbacks from being processed during its execution to prevent race conditions or deadlocks should use the *sys/kn/callback/setflag* function to prevent callbacks being processed.

## 13.3 Kernel Callback Manipulation Functions

### 13.3.1 Set a callback - sys/kn/callback/set

This function sets a callback for the process whose PID is specified in the callback data structure. The next time the specified process calls *sys/kn/proc/sleep* or *sys/kn/proc/deschedule*, if callbacks are enabled, the relevant callback function will be executed. Callbacks are enabled or disabled depending upon the status of the callback flags, which are described in greater detail in the account of the function *sys/kn/callback/setflag*.

If the target process is currently in the SLEEP state and its callback flags are set to either 0 or 2, it is woken, and EINTR is returned from *sys/kn/proc/sleep*. If the target process is in the SLEEP state and its callback flags are set to 0, the callback is processed immediately in the context of the target process, before it returns from *sys/kn/proc/sleep*.

The CALLBACK_DATA, CALLBACK_HANDLER and CALLBACK_PID fields of the data structure should be filled in by the caller before *sys/kn/callback/set* is called. From the time that the callback is set until the callback has either been processed or abandoned, the memory allocated for this structure must remain reserved for this purpose.

It is the responsibility of the process in whose context the callback is executing to ensure that the callback structure passed in is freed when it is no longer required. This tends to be a simple to arrange since this process is usually the same as that which originally allocated the callback structure.

This function may be called from within an interrupt handler or from normal process context.

### 13.3.2 Unset a callback - sys/kn/callback/unset

This function unsets the specified callback, so that it cannot be processed at a future point. The caller must ensure that the specified callback has been set and is still waiting to be processed. If the specified callback is not currently pending, the behaviour is undefined. Once the callback has been unset, the data structure must be freed if it is no longer required.

### 13.3.3 Disable/Enable callbacks for the calling process - sys/kn/callback/setflag

This function enables or disables callback processing for the calling process, depending on the value of the input flag.

If the value of the input flag is 0, callbacks are enabled. In this state, if a callback is pending at the time that *sys/kn/proc/sleep* is called, or becomes pending during the sleep, this will cause *sys/kn/proc/sleep* to return EINTR, and cause the callback to be taken before the sleep returns. If *sys/kn/proc/deschedule* is called, any pending callbacks will be taken immediately.

If the value of the flag is 1, callbacks are completely disabled, and a callback being pending or becoming pending will not cause *sys/kn/proc/sleep* to return. Callbacks will not be taken on a call to *sys/kn/proc/deschedule*.

If the value of the flag is 2, a callback will cause a call to *sys/kn/proc/sleep* operation to return EINTR, but will not cause the callback to be taken. In this flag state, callbacks will not be taken on a call to *sys/kn/proc/deschedule*.

### 13.3.4 Process any pending callbacks - sys/kn/callback/process

This function processes all the current process' outstanding callbacks. An explicit call to this function overrides the callback disable flag state, and thus any pending callbacks are processed even if callbacks are 'disabled.'

### 13.3.5 Returns the value of the PF_CALLBACK_OCCURRED flag for the calling process - sys/kn/callback/occurred

The value returned by this function is the current value of the flag PF_CALLBACK_OCCURRED. This indicates whether a callback has occurred for the calling process since the tool *sys/kn/callback/clr_occurred* was last called to clear the flag. A value of 1 indicates that a callback has occurred, 0 that none has.

It should be noted that *sys/kn/proc/sleep* implicitly clears this flag. The flag is also modified by *sys/kn/callback/process* and *sys/kn/callback/clr_occurred*.

### 13.3.6 Clears the PF_CALLBACK_OCCURRED flag for the calling process - sys/kn/callback/clr_occurred

This function clears the PF_CALLBACK_OCCURRED flag for the calling process. Subsequent calls to *sys/kn/callback/occurred* will return a value indicating that no callbacks have been processed, until a call (either explicit or implicit) to *sys/kn/callback/process* processes a pending callback and sets the flag again.

### 13.3.7 Indicate whether a callback is currently pending for the calling process - sys/kn/callback/pending

This function returns a value indicating whether any callbacks are pending for the calling process. A non-zero value indicates one or more callbacks pending, a zero value that there are none.

# 14. Named Data Area Functions

The intent kernel provides the facility for processes to associate data pointers with strings.

A Named Data Area (NDA) can be accessed by an unlimited number of processes running on the same processor. These processes, which may be of any type, access the NDA by calling the kernel function *sys/kn/atom/find* to look up the name string in the NDA table, and return the associated pointer. Thus it is possible to implement processor-wide data structures that may be shared between many processes.

### 14.1.1 Associate the specified pointer with the specified string - sys/kn/nda/name

This function creates a record of the association between the specified data area pointer and the given string, and stores the record in a table. Since this record is within the kernel structure, it is accessible by all processes running on the same processor. Other processes, therefore, can look up the name, and access the corresponding data area pointer.

If there is already an NDA with the specified name, the data pointer corresponding to that name is returned. If the function successfully creates a new record, the pointer to the input data area is returned. Thus, comparing the input and output data pointers provides a means of checking whether an NDA with the chosen name already existed.

Passing in a NULL data pointer to the function will successfully set up an NDA associated with that address. However, a subsequent successful attempt to find this NDA name will be indistinguishable from a failure, since NULL will be returned from *sys/kn/nda/find* in both cases. It is therefore not usually advisable to set up an NDA with a NULL pointer.

### 14.1.2 Delete the NDA record for the specified string - sys/kn/nda/del

This function deletes the NDA record for the specified string if it is found, and returns 0. If the string is not found, EINVAL is returned. Any subsequent attempt by a process to find the data pointer by using *sys/kn/nda/find* will fail.

The *sys/kn/nda/del* function only removes the relevant string and associated data pointer from the table in the kernel data area. It does not affect the data area itself. Thus, any process which already has the data pointer may continue to use it.

It should be noted that the pointer to the NDA data may already be held by one or more processes. If the data memory corresponding to the NDA is to be freed or reused, the application must first co-operate with any other processes using the NDA to find some way to ensure that the memory is unused.

### 14.1.3 Look up the NDA record for the specified string - sys/kn/nda/find

This function looks up the specified string in the NDA table. If a record containing the specified string is found, the corresponding data area pointer is returned. If no record is found for the specified string, NULL is returned.

# 15. Atoms

While a string of characters is more easily recognised and interpreted by the programmer, machines operate more efficiently and rapidly when working with numbers. It is often useful to represent a string in a more compact form, while retaining its uniqueness.

For this purpose, int**e**nt uses atoms. These are integer values that each uniquely represent a particular string. The int**e**nt kernel provides functions to create, delete and look up atoms. Each processor in an int**e**nt system possesses an atom table, which records the mapping between integers and the corresponding strings.

The int**e**nt kernel stores atoms in different ways, depending upon whether they are created statically by the Sysgen utility, or dynamically by the kernel itself.

## 15.1 Atom values

Atom values are 32-bit values, composed of two parts. The most significant 24 bits is a unique identifier assigned at run-time by the system (or by sysgen, in the case of statically generated atoms). This value cannot in general be predicted in advance. The least significant 8 bits of the value are found by adding together the bytes composing the string that the atom represents, and using the bottom 8 bits of the sum. This is always constant for a given string, and so can be used at assembly time to implement a simple hash function.

## 15.2 Static Atoms

Static atoms are stored in a table created by sysgen. The atoms are stored in such a way that both name to value lookups and value to name lookups can be done using a binary-chop algorithm, resulting in very high performance. In addition, the textual data for static atoms may be compressed to save space in the image.

Atoms that are statically generated exist for the lifetime of the system, and may even be placed in ROM so that they cannot be modified. They are not reference counted, and do not cease to exist when unused by the system. They require less memory than dynamically generated atoms, and they may be compressed to a higher level without requiring much run-time overhead.

## 15.3 Dynamic Atoms

Dynamically generated atoms are stored either in a singly linked list or in an AVL tree, depending on the configuration of the system. The AVL tree version is considerably faster, but has a higher memory overhead.

## 15.4 Atom functions

### 15.4.1 Return atom value for specified string, create new atom if required - sys/kn/atom/add

This function searches for an atom representing the specified string. If such an atom is found, the corresponding atom value is returned. If no such atom is found, it is dynamically created. The specified string is added to the system's atom table, and the function returns the corresponding atom value, which may subsequently be used to identify the string. The reference count of the new atom is initialised to 1.

Once created, the reference count of a dynamic atom is incremented by any subsequent calls to *sys/kn/atom/add* that specify the same string. The same atom value is returned in these cases. The atom will continue to exist until its reference count drops to zero. The reference count is decremented when *sys/kn/atom/del* is called.

If an error occurs, zero is returned. Zero is not a valid atom value.

### 15.4.2 Dereference the specified atom, delete if unreferenced - sys/kn/atom/del

If the atom specified is a dynamic atom, this function decrements the reference. If the reference count is decremented to zero, the atom is removed from the atom table. Once this happens, the specified atom value may no longer be validly used to refer to the previously associated string.

If the same string is added to the atom table again, the atom value returned may be different.

If the specified atom value corresponds to a statically generated atom, no action is taken and success is returned.

### 15.4.3 Return the atom value corresponding to the specified string - sys/kn/atom/find

This function looks up the specified string in the atom table and returns the corresponding atom value. If there is no atom for the specified string, zero (an invalid atom value) is returned.

The reference count of the atom is not incremented, even if it is successfully found.

### 15.4.4 Return a copy of the string corresponding to the specified atom - sys/kn/atom/getname

This function looks up the atom specified by the given atom value. If found, the corresponding string is copied into the buffer provided by the user.

If the buffer pointer specified is NULL, the string is not copied, but the size of the buffer required to contain the string is returned.

### 15.4.5 Increment the reference count of an atom - sys/kn/atom/ref

This function looks up the specified atom. If it is found, the reference count is incremented.

# 16. Kernel Notification Functions

intent supports a mechanism which allows applications, device drivers, debuggers and so forth to subscribe to various system events so as to be notified when they occur.

Notification takes the form of a call to a function registered by the subscriber. This call can take place through one of a number of mechanisms. For example, it can take place in the context of the process which registered the subscription, exactly like a callback. Alternatively, the call can occur in the same context as whatever generated the relevant system event. The mechanism to be used is determined at subscription time, and may be different for each subscriber.

Two data pointers are passed to the handling function. One of these points to a Notify Data Structure which stores data about the event as well as information used by the callback mechanism. A description of this structure may be found later in this document in the section on data structures. The other pointer indicates data supplied by the subscriber at subscription time. This may contain data of any sort, since it will only be used by the subscriber's handling function.

It is possible for a process to be subscribed invariably to certain events, if they are a standard part of the system. However, new events can also be added dynamically. If the event requested by the subscriber is not a standard system event, this will take place automatically. Subscriptions are normally made by means of an event ID code, or optionally by means of a name string. The event ID codes are derived from the names using the kernel atom functions. For more details of these, please consult the earlier section on atom functions.

Where the notification is to be handled in the subscriber's context, a callback data structure is linked to the process's callback list as it would with any other callback. Where notification is handled within the context of the generator, a callback structure is still used to store the subscriber data pointer and the pointer to the handling function, but is not linked to the generating process's callback list.

## 16.1 System Events

Certain event names are reserved and correspond to pre-defined system events. The meaning of the NOTIFY_EVENT_DATA pointer is specific to each event type. This section provides some examples of likely uses of the notification mechanism.

| Event Name | Event Data |
|---|---|
| "KernelEvent_ToolLoad" | Pointer to array of tools |
| "KernelEvent_ToolFlush" | Pointer to array of tools |
| "KernelEvent_ToolAnnul" | Pointer to tool |
| "KernelEvent_AtomAdd" | Pointer to atom number |
| "KernelEvent_AtomDel" | Pointer to atom number |

## 16.2 The event handling function

This is the function specified as the event handler by the first parameter to *sys/kn/notify/subscribe*, and conforms to the interface described above.

When a request is made for notification of a particular event, a pointer to a handling function is provided. This handling function is called when the event occurs via one of a number of mechanisms as selected by the original call to *sys/kn/notify/subscribe*. Two pointers are passed to it, both of which

are provided by the subscriber at subscription time. The NOTIFY_EVENT_DATA field of the Notify Data Structure is filled in when the event is generated, not by the subscriber.

If the callback is occurring in process context, it is the responsibility of the handler to free the memory used for the callback structure. The notification mechanism cannot free the memory, as it cannot guarantee not to do so before the callback has been taken. Furthermore, it cannot be freed by the callback mechanism, since this behaviour is not desirable for other uses of callbacks.

## 16.3 Kernel Notification Functions

### 16.3.1 Announce that an event has taken place - sys/kn/notify/announce

This tool notifies all interested parties, namely those which have previously called *sys/kn/notify/subscribe*, that a particular event has been generated. It is the responsibility of the caller, before calling this tool, to ensure that the correct event data is indicated by the event data pointer.

The event handle passed in to this function should be a value returned by either *sys/kn/notify/generator* or *sys/kn/notify/generator_name*.

### 16.3.2 Register an interest in being notified when a particular event occurs - sys/kn/notify/subscribe

This tool allows a process to subscribe to an event using a unique ID code. Thereafter, whenever this event is announced by means of *sys/kn/notify/announce* then the specified handling function will be called. The handling function need not always be a straightforward callback and further mechanisms may be added in future.

An integer parameter specifies whether the handling function is to be executed within the context of the process generating the event, or that of the subscriber process. Choice of which context to use may be dictated by, for example, requirements to access of the global pointer of one or other of the processes.

Subscribing to a previously unknown event type will cause that event type to be dynamically created.

If subscribing to a callback in process context, it must be remembered that, as with other callbacks, the callback data structure must be freed when it is no longer required. The data structure is passed as a parameter to *sys/kn/notify/announce*, and must be freed by the process in whose context the callback is being taken.

### 16.3.3 Register an interest in being notified when a particular named event occurs - sys/kn/notify/subscribe_name

This tool allows a process to subscribe to an event in a fashion identical to *sys/kn/notify/subscribe*, but using a string to refer to the event, rather than an integer acting as an ID code.

The textual name of the event (specified by the string) may be a name previously passed to *sys/kn/notify/generator_name*, or may be an unknown event type which will be dynamically generated. The 'notification mechanism' parameter controls what form the notification will take. This need not always be a straightforward callback.

Subscribing to a previously unknown event type will cause that event type to be dynamically created.

If subscribing to a callback in process context, it must be remembered that, as with other callbacks, the callback data structure must be freed when it is no longer required. The data structure is passed

as a parameter to *sys/kn/notify/announce*, and must be freed by the process in whose context the callback is being taken.

### 16.3.4 Withdraw previous subscription to be notified when a certain event is generated - sys/kn/notify/unsubscribe

This tool cancels an earlier subscription to an event. The token returned when interest in the event was registered (using either *sys/kn/notify/subscribe,* or *sys/kn/notify/subscribe_name)* is passed as a parameter to identify which subscription is to be removed. If after this the event type has no subscribers or generators, it is deleted.

### 16.3.5 Register as a generator of a particular kind of event - sys/kn/notify/generator

This tool informs the kernel of a new generator for a particular kind of event. A handle is then returned which is passed to *sys/kn/notify/announce* whenever the event is generated. Registering a generator for a previously unknown event type will cause that event type to be dynamically generated.

The user must establish a policy for the allocation of ID codes to identify the event, and for this reason may find it easier to use *sys/kn/notify/generator_name.*

### 16.3.6 Register as a generator of a particular kind of named event - sys/kn/notify/generator_name

This tool informs the kernel of a new generator for a particular kind of event. This function behaves in the same way as *sys/kn/notify/generator*, except that the event is identified using a textual name, represented by a string, rather than an ID code.

### 16.3.7 Cease to be a generator of a particular type of event - sys/kn/notify/ungenerator

This tool unregisters an event generator, i.e. records that a particular process no longer wishes to be a generator for this event. If after this the event type has no registered generators or subscribers, it is deleted.

The token used to identify the particular event generator should be a value returned by either *sys/kn/notify/generator* or *sys/kn/notify/generator_name.*

# 17. Atomic List Functions

Alterations to a linked list cannot be interrupted without the risk of leaving the list in a temporarily 'broken' state, which may cause undefined behaviour in any operation that attempts to use it. Therefore, it is often necessary to add or remove items from a list in an atomic manner, so that the operation is indivisible and may not be halted before completion.

The Platform Isolation Interface (PII) provides tools for switching interrupts on and off. These can be used to ensure that interrupts cannot halt list modifications before completion. Details of the functions usually used for interrupt handling may be found in the PII documentation.

The functions described below, however, provide a somewhat more convenient interface for altering lists in an atomic manner. These functions are atomic with respect to interrupts and all other forms of context switching.

### 17.1.1 Add a node to the head of the list - sys/kn/atomic/addhead

This function adds the specified node at the head of the indicated list. The caller process cannot be interrupted while the list operation is in progress, so other processes and interrupt handlers cannot access the temporarily broken list.

### 17.1.2 Add a node to the tail of the list - sys/kn/atomic/addtail

This function adds the specified node at the tail of the indicated list. The caller process cannot be interrupted while the list operation is in progress, so other processes and interrupt handlers cannot access the temporarily broken list.

### 17.1.3 Add a node after the specified list node - sys/kn/atomic/addnode

This function places the node pointed to by the second pointer parameter on a list, after the node specified by the first. Since a node must not be linked to more than one list, there is no need for a parameter specifying the list.

The return value is a pointer to the node which is immediately after the newly added node.

The caller process cannot be interrupted while the list operation is in progress, so other processes and interrupt handlers cannot access the temporarily broken list.

### 17.1.4 Add a node before the specified list node - sys/kn/atomic/addnodeb

This function places the node pointed to by the second pointer parameter on a list, before the node specified by the first. Since a node must not be linked to more than one list, there is no need for a parameter specifying the list.

The return value is a pointer to the node which is immediately before the newly added node.

The caller process cannot be interrupted while the list operation is in progress, so other processes and interrupt handlers cannot access the temporarily broken list.

### 17.1.5 Remove the node from the head of the specified list - sys/kn/atomic/removehead

This function removes a node from the head of the specified list. It then returns a pointer to the removed node. The caller process cannot be interrupted while the list operation is in progress, so other processes and interrupt handlers cannot access the temporarily broken list.

If there are no nodes in the list, NULL is returned.

### 17.1.6 Remove the specified node from its list - sys/kn/atomic/removenode

This function removes the specified node from the list containing it. Since a node must not be linked to more than one list, there is no need for a parameter specifying the list.

The return values indicate the nodes which were immediately before and immediately after the removed node. It should be noted that since *sys/kn/atomic/removenode* returns two values the multiple return registers facility of the Elate C compiler should be used if calling this tool from C.

The caller's process cannot be interrupted while the list operation is in progress, so other processes and interrupt handlers cannot access the temporarily broken list.

### 17.1.7 Moves the entire contents of one list onto another - sys/kn/atomic/movelist

This function moves all of the nodes on one list onto another. This operation is performed atomically and deterministically, and the time spent performing this operation is not dependent on the number of nodes in either list. A flag can be set to indicate whether the nodes are to be added to the head or the tail of the destination list.

This operation is atomic with respect to interrupts and all other forms of context switching.

# 18. Mini Atomic Blocks

Mini atomic blocks (MABs) are a mechanism for ensuring the atomicity of certain operations with respect to each other. An operation is said to be atomic with respect to a set of things if none of them can occur during the execution of the operation.

Different kinds of MABs exist to exclude different kinds of things. Whilst execution is within the scope of a MAB, the excluded things are guaranteed not to occur.

MABs are nestable, since it is possible to use a MAB within the scope of another MAB.

### 18.1.1 Enter a scheduler MAB - sys/kn/mab/begin_s

This tool begins a scheduler MAB. Within the scope of a scheduler MAB, no scheduler activity may take place. This prevents any thread other than that currently running from executing, regardless of priority. It will also prevent timer processing. Interrupts may still occur, however, which may result in the execution of interrupt handler routines.

If a process goes to sleep whilst within a scheduler MAB (or, equivalently, blocks whilst waiting on some synchronisation object) then this will allow other threads to schedule. However, once the sleeping process is woken and runs again, it will resume execution within the MAB, so scheduling will once again be prevented.

The return value is not defined to have any particular form. It must be passed in to *sys/kn/mab/end_s* at the end of the MAB's scope.

### 18.1.2 Leave a scheduler MAB - sys/kn/mab/end_s

This tool marks the end of a scheduler MAB. The value passed into this tool must be the value returned by *sys/kn/mab/begin_s*.

### 18.1.3 Enter a thread MAB - sys/kn/mab/begin_t

This tool begins a thread MAB. Within a thread MAB, no other threads besides the current thread within the calling process may run. However, threads within other processes may run as normal.

The return value is not defined to have any particular form. It must be passed in to *sys/kn/mab/end_t* at the end of the MAB's scope.

Since intent does not currently have a notion of threads as distinct from processes, thread MABs behave identically to scheduler MABs. However, any thread implementations which may exist in future will behave correctly with respect to thread MABs and so these should be used if and only if they are appropriate.

### 18.1.4 Leave a thread MAB - sys/kn/mab/end_t

This tool marks the end of a thread MAB. The value passed into this tool must be the value returned by *sys/kn/mab/begin_t*.

### 18.1.5 Enter an interrupt MAB - sys/kn/mab/begin_i

This tool begins an interrupt MAB. Within an interrupt MAB, interrupts are disabled and cannot occur. This also prevents pre-emptive descheduling of the current thread.

The return value is not defined to have any particular form. It must be passed in to *sys/kn/mab/end_i* at the end of the MAB's scope.

### 18.1.6 Leave an interrupt MAB - sys/kn/mab/end_i

This tool marks the end of an interrupt MAB. The tool is passed a value which indicates the MAB state which should be restored. This value must be the value returned by *sys/kn/mab/begin_i.*

# 19. AVL Tree Management Functions

The kernel contains functions for the management of AVL trees within intent. AVL trees are a specific form of balanced binary tree.

A binary tree is a tree graph, each node of which has at most two outgoing edges. Balanced binary trees are structured so as to limit imbalances in size between two subtrees of any node.

A balanced binary tree is an efficient general purpose data structure. The primary advantage of this structure is that it is possible to add, remove and search for elements in such a way that, as the size of the tree increases, the time required only increases in proportion to the log of the number of elements in the tree. This is a significant improvement over linked lists, which require time linear in the length of the list for operations involving searching, though addition and removal of nodes is only of order 1.

AVL trees (named after Adelson-Velskii and Landis, the inventors of the system) are a type of balanced binary tree.The criterion for balance at a node of an AVL tree is that the difference in the height of the two subtrees is never more than one. Height and depth for trees are defined as follows:

- The height of a tree with no elements is 0.
- The height of a tree with one element is one. The depth of the root node of any tree is 1.
- The height of a tree with more than one element is the height of the tallest subtree plus one. The depth of a node in such a tree is the depth of its parent, plus 1.

An example of a tree that fits this criterion is shown below.

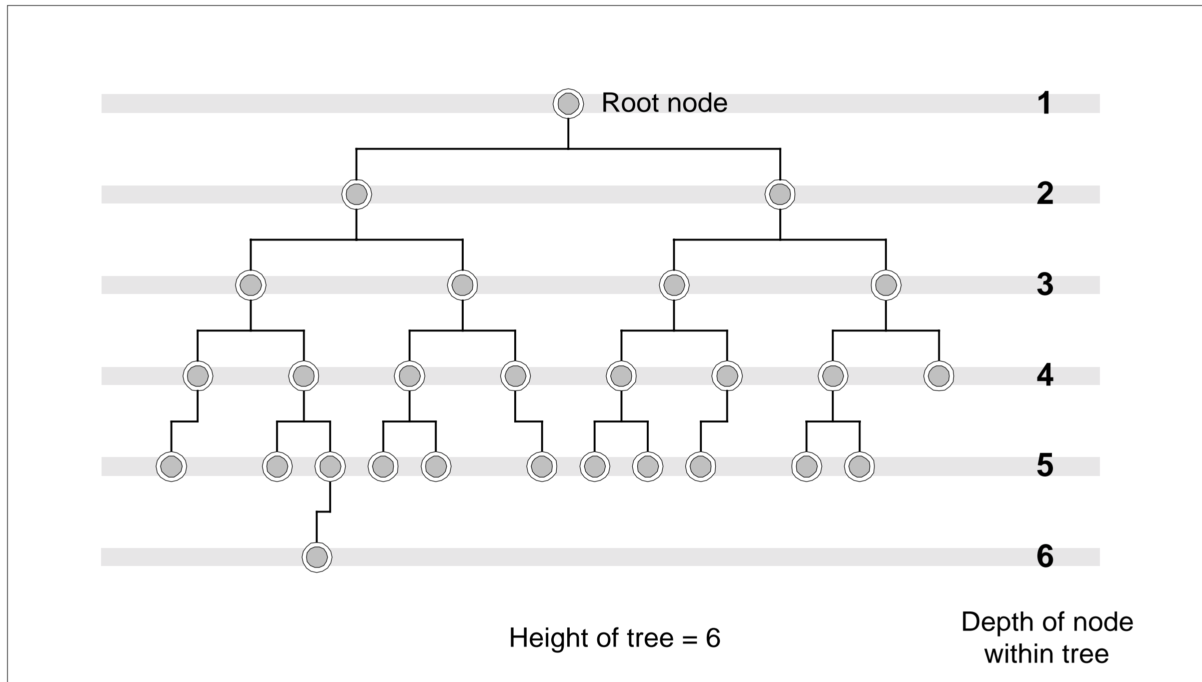


**Diagram 9 - The AVL Tree**

The 'balanced' property of an AVL tree is maintained incrementally in a time-efficient manner, as previously described. Whenever a node is inserted or removed, one or more 'rebalancing' transformations are performed upon the tree.

The int**e**nt implementation of AVL trees is non-recursive and therefore does not rely upon the allocation of memory or availability of stack space.

In order to implement the functionality required, AVL trees use comparison routines to order the data they contain. If there is no way to define a well-ordering of the data, then it is not suitable for use with an AVL tree.

Each AVL tree has a main structure which holds a pointer to the root element of the tree, and pointers to the comparison routines which order the elements within the tree. The fields of this structure should only ever be changed using the tools defined in this section during normal operations on the AVL tree. In addition, each element in the tree requires a separate structure at the same offset from the beginning of the element.

As with most int**e**nt procedures, it is the responsibility of the application to allocate memory for tree headers and nodes, and to free it when it is no longer required. Removing a node from a tree (using *sys/kn/avl/remove* or *sys/kn/avl/removeall*) does not free the memory associated with it.

It should be noted that int**e**nt AVL trees have no inherent concurrency control. If there is a possibility of several threads or process accessing the tree concurrently, some external means must be used to guarantee that operations which modify the tree are performed in an atomic manner. Such operations would include the insertion and removal of nodes. It is the caller's responsibility to ensure this, though the Elate kernel provides a variety of mechanisms to achieve this goal.

## 19.1 Kernel Management Functions for the AVL Tree

### 19.1.1 Initialises an AVL tree header - sys/kn/avl/init

This tool initialises the AVL tree header indicated by the first input pointer. No elements are added, and the tree is left empty.

As previously stated, the AVL node structures of a tree are each placed at the same offset from the start of the element in which they are embedded. The input integer contains the value of this byte offset for each node that will be added to the tree. This offset must be a multiple of 4. It should be noted that the AVL tree initialised by this tool will not permit the insertion of elements with duplicate key values. If such elements are required then *sys/kn/avl/init_dup* should be used instead. If duplicate values are inserted into the tree, *sys/kn/avl/init* will return an error.

The second input pointer indicates the comparison function to be used for the tree. This function takes pointers to two elements of a tree, and returns a value which is less than, equal to or greater than zero to indicate whether the first element is less than, equal to or greater than the second.

### 19.1.2 Initialises an AVL tree header with relocatable stack - sys/kn/avl/init_rs

This tool performs the same function as *sys/kn/avl/init* except that the stack used as workspace by the AVL algorithms is passed in explicitly by the user via the third pointer parameter.

The block supplied by the caller must be at least *avl_kstack_size* bytes in length (see the description of the AVL tree structure in the later section on "Data Structure Definitions"). It will only be used by the AVL algorithms during calls to AVL functions and so under some circumstances can be used for other purposes when no such calls are taking place.

It is permissible to pass in a NULL stack. The tree will then be unusable until the *sys/kn/avl/move_rs* tool is called passing in a usable stack.

### 19.1.3 Initialises an AVL tree header to permit elements with duplicate keys - sys/kn/avl/init_dup

This tool initialises the AVL tree header indicated by the first input pointer, in a manner similar to *sys/kn/avl/init*. However, the tree resulting from use of this function does permit the insertion of elements with duplicate keys. This is achieved through the use of a second comparison function, which provides a means of ordering two elements with duplicate keys.

The third input pointer points to this 'tie-breaker' comparison function, which refines the comparison function indicated by the second input pointer. Where the standard comparison function returns a non-zero value, the tiebreaker comparison function will return the same value. Whenever the value returned by the standard comparison function is zero, indicating that the keys of the elements are identical, the tiebreaker function instead imposes a consistent ordering between the two elements. This ordering may be achieved by, for example, comparing the values of the pointers to the elements.

This stronger comparison is used for insertion and removal of elements. Standard comparison is used for other purposes, such as searching the tree. Both comparison functions have the form described under *sys/kn/avl/init*.

### 19.1.4 Initialises an AVL tree header with relocatable stack to permit elements with duplicate keys - sys/kn/avl/init_dup_rs

This tool performs the same function as *sys/kn/avl/init_dup* except that the stack used as workspace by the AVL algorithms is passed in explicitly by the user via the fourth input pointer parameter.

The block supplied by the caller must be at least *avl_kstack_size* bytes in length (see the description of the AVL tree structure in the later section on “Data Structure Definitions”). It will only be used by the AVL algorithms during calls to AVL functions and so under some circumstances can be used for other purposes when no such calls are taking place.

### 19.1.5 Switch the location of the relocatable stack - sys/kn/avl/move_rs

This tool switches to a new location for the AVL workspace stack. It is passed two parameters, one of which points to the header of the relevant AVL tree, and one of which is may be NULL, may point to the new stack, or may hold a value of -1.

If this latter parameter is NULL, then this tool simply returns a pointer to the old stack. If it is –1, then the tree cannot be used until a valid stack is passed in at a later point. This can be useful where it is desirable to ensure that, for example, an AVL tree stack allocated from a process' stack is not used after the routine in which it was allocated has returned, since this could otherwise have odd side effects.

If the pointer returned by this tool has value -1, this means that no stack was set prior to the call.

### 19.1.6 Deinitialises an AVL tree header - sys/kn/avl/deinit

This tool deinitialises the specified AVL tree header, making further operations on the tree invalid.

### 19.1.7 Inserts a new node into an AVL tree - sys/kn/avl/insert

This tool inserts the specified new node into the indicated AVL tree, which is then rebalanced. The first input pointer indicates the header of the AVL tree into which the element is to be added, and the

second pointer points to the element to be inserted. If an element with the same key already exists in the tree, then the new element is inserted just to the right of the existing one(s) (providing the tree was initialised using *sys/kn/avl/init_dup*). If the specified element itself is already in the tree, behaviour is undefined.

### 19.1.8 Removes a node from an AVL tree, rebalancing the tree - sys/kn/avl/remove

This function removes the given element from the specified tree.

### 19.1.9 Removes all the elements from an AVL tree - sys/kn/avl/removeall

This tool removes all the elements from the specified tree, leaving it empty. For each element in the tree the callback function is called and passed pointers to the user's data and to the current element. If the callback function ever returns a non-zero value then no further nodes will be processed via the callback, the tree will be left empty and the callback function's return value is passed back to the caller of this tool. The order in which the nodes of the tree are enumerated is undefined.

Any attempt by another process to perform a tree operation at the same time will result in undefined behaviour.

This function does not deinit the tree header, and so the user can begin adding nodes to the tree again once *sys/kn/avl/removeall* has returned.

### 19.1.10 Removes elements within the given range from an AVL tree - sys/kn/avl/removerange

This tool removes those elements of the tree the nodes of which lie between two specified dummy nodes in the ordering. This range includes the two specified nodes themselves. Elements are removed in the order defined by the tree's ordering.

For each element removed from the tree a specified callback function is called with pointers to the user's data and the element currently being removed. If the callback function ever returns a non-zero value then the node in question is not removed and the callback is not applied to the remaining nodes, which are also not removed. In this case the callback function's return value is passed back to the caller of this tool. The order in which elements are returned is undefined.

Note that it is not necessary for the nodes specified as the extremities of the range to contain values which actually exist within the tree.

During the execution of this tool any attempt by another thread to perform another operation on the target tree will result in undefined behaviour.

### 19.1.11 Finds an element within a tree - sys/kn/avl/find

This tool finds an element within the specified AVL tree which matches the given key. The second input pointer indicates a structure which resembles an element sufficiently to enable the tree's comparison function to compare it with such an element. This can be a block of memory of the correct size, with the key written into the correct location. The AVL node structure embedded within normal elements is ignored within the key specified by the second input parameter.

If the AVL tree contains more than one element with a matching key, the node returned will be the leftmost.

### 19.1.12 Finds a node with a specified key value within a tree - sys/kn/avl/findkey

This tool finds an element within the AVL tree, the key of which matches a specified key value. It behaves identically to *sys/kn/avl/find* except that a key value and a comparator are passed in rather than a dummy node. The custom comparator must impose the same ordering as the comparison function for the tree. However, it should take as its first parameter a key value rather than a node. In particular, it should be noted that the first parameter to the comparator is still a pointer.

The purpose of this tool is simply to remove the need to construct a dummy node in cases where the copying of the desired key value into the dummy node would be an unacceptable performance overhead.

If the AVL tree contains more than one element with a matching key, the node returned will be the leftmost. Others can be accessed by using *sys/kn/avl/walkright*.

### 19.1.13 Returns the maximal element within an AVL tree - sys/kn/avl/maximum

This tool returns the maximal element within a tree, using the comparison function specified at initialisation. If there is no maximal element because the tree is empty, then NULL is returned.

### 19.1.14 Returns the minimal element within an AVL tree - sys/kn/avl/minimum

This tool returns the minimal element within a tree, using the comparison function specified at initialisation. If there is no minimal element because the tree is empty, then NULL is returned.

### 19.1.15 Enumerates all elements in an AVL tree, calling a function for each one - sys/kn/avl/enumerate

This tool enumerates all elements within the specified AVL tree in a left-to-right order (i.e. minimal to maximal). For each element, the specified callback function is called, and is passed the data pointer originally passed to *sys/kn/avl/enumerate*, and the pointer to the current node. If the callback function returns a non-zero value, then the enumeration is aborted and the same return value is returned by this tool.

### 19.1.16 Returns the number of elements in an AVL tree - sys/kn/avl/size

This tool returns the number of elements in the specified AVL tree.

### 19.1.17 Checks the consistency of an AVL tree - sys/kn/avl/check

This tool checks the consistency of an AVL tree. The checks performed will fail if the input pointer does not point to a valid AVL tree, all the flags and walkpointers of which accurately describe the tree's structure.

When writing an AVL application, it should be noted that a successful return from this tool obviously does not prevent errors where the tree is consistent within itself but nonetheless contains wrong elements.

### 19.1.18 Finds the leftmost element greater than or equal to a key - sys/kn/avl/ubound

This tool finds within the given AVL tree the leftmost element that matches or is greater than the specified key. The second input pointer points to a structure which resembles an element sufficiently

to enable the tree's comparison function to compare it with such an element. The AVL node structure embedded within normal elements is ignored within the key specified by the second input parameter.

If no suitable element exists, since all elements are less than the key, then NULL is returned.

### 19.1.19 Finds the rightmost element less than or equal to a key - sys/kn/avl/lbound

This tool finds the rightmost element within the specified AVL tree which matches or is less than the specified key. The second input pointer points to a structure which resembles an element sufficiently to enable the tree's comparison function to compare it with such an element. The AVL node structure embedded within normal elements is ignored within the key specified by the second input parameter.

If no suitable element exists, since all elements are less than the key, then NULL is returned.

### 19.1.20 Finds the next element left from a given element (ie. next smaller key) - sys/kn/avl/walkleft

This tool finds the next element left from the specified node. Although in the worst case this may take a length of time proportional to the height of the tree, on average it is much quicker. If the starting element is the leftmost element of the given tree, then this tool returns NULL.

### 19.1.21 Finds the next element right from a given element (ie. next larger key) - sys/kn/avl/walkright

This tool finds the next element right from the specified node. Although in the worst case this may take a length of time proportional to the height of the tree, on average it is much quicker. If the starting element is the rightmost element of the given tree, then this tool returns NULL.

# 20. Kernel Device Functions

In int**e**nt, devices are made available to applications through the use of a mount table. Each mount table record contains the device ID, the name of the mount point, and some flags.

The device ID is a 64 bit value, divided into two parts. The least significant 32 bits contains the device driver's instance pointer. The most significant 32 bits contains the number of the processor upon which the device is situated.

The mount point name is assigned when the device is mounted. If the *devstart* program has been used to mount the device, then the mount point will be specified by the user. If the device is mounted when the system boots, then the system-integrator will specify the mount point. If different names are associated with the same pointer, then more than one instance of a device driver may exist in memory at the same time.

The flags are also specified at the time the device is mounted. These can be used to specify whether the device is network-visible, or only visible to processes on the same CPU. Link drivers and similar device drivers will usually be mounted to be locally-visible only.

The int**e**nt kernel provides a range of functions for manipulating the device mount table, and for looking up specific devices within it.

### 20.1.1 Look up a device in the system mount table - sys/kn/dev/lookup

This function looks up the specified device in the mount table and returns details of the best match found.

In many cases, there may be an incomplete match. For example, systems often have a filesystem device mounted as "", which will match any name. However, if the mount table also contains a filesystem mounted as "*var*" a lookup on the name "*var/tmp/tmpfile0001"* will return the "*var*" filesystem instead of "".

In addition to the device object instance pointer, a string pointer is returned. This is a pointer to the end of the matched string within the input string (with any leading slash stripped off). In the above example, the string would be “tmp/tmpfile0001”. This represents the filename relative to the root of the matched device.

If an exact match is required, the caller should check that returned string pointer points to a nul char.

If the device exists on a different processor from that upon which the lookup is taking place, a local alias for the device will be created, and a pointer to this object returned. The alias is designed to form a transparent interface between application and driver. The application is able to send an *ncall* to the alias, as if to a device driver local to the application's own processor.

An "agent" process, local to the real device object, then retrieves information concerning the parameters of the *ncall*. The agent performs the same *ncall* to the device driver, which treats it as if it had been sent by a local application. The results of the *ncall* are then sent by the agent to the alias. The agent process and the alias are created only when the remote device is first accessed.

If the mount table entry indicates that the device was mounted using the delayed-mount option, then, before returning the device ID, this function will instigate the execution of the normal sequence of operations for initialising a device.

This entails loading the *_new* tool to create the object. After this, the *init* method is called using parameters specified in the corresponding call to *sys/kn/dev/mount_delayed*. If these initialisation steps are successful, this function returns the object ID to the caller in the usual manner.

### 20.1.2 Look up a device in the system mount table - sys/kn/dev/rlookup

This function looks up the specified device in the mount table and returns the corresponding device name string.

### 20.1.3 Add a device to the system mount table - sys/kn/dev/mount

This function inserts the specified device ID into the mount table, giving it the specified name. The device is 'mounted,' placed into a hierarchy of devices. This operation renders the device accessible to any processes that seek it by name in the mount table.

At present, the only flag defined is the MNTF_LOCAL flag. If this flag is set, the device is not visible to processes on CPUs other than that upon which it is mounted.

### 20.1.4 Remove the specified device from the system mount table - sys/kn/dev/unmount

This function removes the specified device from the mount table. This does not prevent the device being accessed by processes which already have its device object instance pointer, but processes attempting to look up the device by name will no longer be able to find its instance pointer.

### 20.1.5 Adds a delayed-mount record for a device to the mount table - sys/kn/dev/mount_delayed

This function stores details of an association between a mount point and a specific parameter by creating a record in the mount table. This record contains details of the mount point, and the parameters of the relevant device. The device in question is considered to have been mounted using the 'delayed-mount' option.

When the function *sys/kn/dev/lookup* is used to reference the device, the parameters in this record are used transparently to initialise the device, before the device ID is returned in the normal manner.

### 20.1.6 Adds a new device driver to a running system - sys/kn/dev/start

This tool starts a new device driver whilst the system is running. To load device drivers from the sysgen steering file, the *.obj* directive should be used. A new device driver object belonging to the specified class is created, and then added to the mount table associated with the specified path.

There are currently two option flags which can be set. Bit 0 indicates that the device should be mounted only on the local processor rather than other processors being notified. If bit 1 is set, then a delayed mount should be used (see *sys/kn/dev/mount_delayed*).

Two integers are returned by the tool. The first indicates whether an error has occurred and, if so, at which stage. The second integer is only relevant if an error has occurred, and provides more information specific to the stage at which the tool failed. The different values of the second integer signify the following:

- Stage 0 - Success (no error) - Error value is undefined.
- Stage 1 - Device _new tool open failed - Error value is undefined.
- Stage 2 - Call to new failed - Error value is ENOMEM.
- Stage 3 - Call to init failed - Error value is an error code.

- Stage 4 - Mount failed - Error value is an error code.
- Stage 5 - Out of memory - Error value is ENOMEM.

It should be noted that since this tool returns two values, if calling it from C it will be necessary to use the multiple return registers facility of the Elate C compiler.

### 20.1.7 Unmounts and stops a device - sys/kn/dev/stop

This function removes the specified device from the mount table and then causes it to stop. Care must be taken in calling this tool, since other processes may still have the device object's instance pointer and may not be aware that the device has been stopped.

# 21. Static Areas Support

Normally, the only areas that can be easily referenced by a tool are those addressed directly or indirectly via their parameters, via the PROC structure, or in named data areas. The static areas support allows processes to access per process data rapidly. This is used, for example, in the C shared libraries.

The statics areas support described in this section is dedicated to the rapid retrieval of the addresses of static areas. It is not responsible for the allocation and deallocation of static areas, their format or their contents. The simplest way to set up statics is by using the Simple Statics interfaces.

## 21.1 Static Data Caches

A particular static area, which has a different per process copy for each process using it, requires a four word cache. This cache is typically placed in a tool's writable data. The tool's reference count is incremented while the statics are in use to ensure that the tool, and hence the writable data, does not disappear. A cache is structured as follows:

| Field Type | Field Name | Description |
|---|---|---|
| Pointer | WDA_CACHE_DATAPTR | Static data area pointer |
| Pointer | WDA_CACHE_CHECK | Check – must be read after data pointer |
| Structure | WDA_CACHE_NODE, LN_SIZE | Chain for invalidating |
| Size | WDA_CACHE_SIZE | 16 bytes |

## 21.2 Accessing statics

A static area is accessed as follows:

1. Get the address of the cache. This is typically the tool's writable data, arh ARH_WRDATA.
2. Load the cached static area pointer from WDA_CACHE_DATAPTR (offset 0) in the cache.
3. If the pointer at WDA_CACHE_CHECK (offset 4) in the cache is equal to the global pointer register, then the static area pointer is the correct one.
4. Otherwise, call *sys/kn/statics/statics_get*, which either finds the applicable static area, or creates and initialises a new one. In either case the area's pointer is stored back in the cache.

Step (2) must be performed before step (3), otherwise another process might change the cache in between the two steps. This could cause this code to assume incorrectly that it had been given the static area pointer for the correct thread.

**Example**

The following example shows the simple statics interface in use. A circular buffer is implemented for strings to output when the program ends (i.e. when the static areas are automatically deinitialised).

```
; Set up statics structure e.g. a circular trace buffer

include ‘taort’
include ‘sys/kn/statics/statics.inc’

SPACE_SIZE = 128   ; power of 2

structure
     int32    CBUF_INDEX
```

```
          struct    CBUF_BUF, SPACE_SIZE
          size      CBUF_SIZE

     ; cache for loader object pointer
     .headerext TOOLHDREXT_WRDATA
          dc.ni WDA_CACHE_SIZE    ;16
          dc.b  $ff               ;fill with -1's
     .headerextend

     tool 'testdir/ctrace'

          ent p0 : -
          defbegin 0
          defp pstr
          defp pcache, pspace, pbuf
          defi index, c
          cpy.p arh ARH_WRDATA, pcache
          cpy.p [pcache+WDA_CACHE_DATAPTR], pspace               ;+0
          bcn.p [pcache+WDA_CACHE_CHECK] != gp, out_of_line_get  ;+4
     out_of_line_get_done:
          ; the work of the routine using pspace
          cpy [pspace+CBUF_INDEX], index
          cpy pspace+CBUF_BUF, pbuf
          repeat
               cpy.b [pstr], c
               cpy pstr+1, pstr
               if c == 0
                   cpy.b '\n', [pbuf+index]
               else
                   cpy.b c, [pbuf+index]
               endif
               cpy (index+1) & (SPACE_SIZE-1), index
          until c == 0
          cpy index, [pspace+CBUF_INDEX]
          ret

     out_of_line_get:
          gos get_space, (- : pspace)
          go out_of_line_get_done
          defend 0

     get_space:
          ent - : p0
          defbegin 0
          defp pspace
          defp pcache
          defi errno
          cpy.p arh ARH_WRDATA, pcache
          qcall sys/kn/statics/statics_get, (pcache, __thistool.p,
setup_loader.p, 0.p : pspace, errno)
          if pspace == 0
               tracef "statics_get error %d\n", errno
               qcall lib/abort, (-:-)
          endif
          ret
          defend 0

          .data 4
     setup_loader:
          dc.p __thistool     ; tool pointer
          dc.i CBUF_SIZE      ; size of statics
          dc.p new_space      ; new/alloc
          dc.p 0              ; init
```

```
        dc.p 0              ; wait
        dc.p deinit_space   ; deinit
        dc.p 0              ; delete
        .code

    new_space:
        ent p0 p1 p2 : p0 i0
        defbegin 0
        defp pspace, pstatics, puser
        defi errno
        cpy 0, errno
        ; any special initialisation e.g. initialise buffer
        cpy.i 0, [pspace+CBUF_INDEX]
        qcall lib/memseti, ((pspace+CBUF_BUF).p, 0.i, SPACE_SIZE.i : p~)
        ret
        defend 0

    deinit_space:
        ent p0 p1 : i0
        defbegin 0
        defp pspace, pstatics
        defi errno
        defp pbuf
        defi len, index, limit, cnt, c, lastc
        cpy 0, errno
        ; any special deinitialisation - e.g. dump buffer
        cpy [pspace+CBUF_INDEX], index
        cpy pspace+CBUF_BUF, pbuf
        cpy index, limit
        cpy 0, lastc
        tracef "cbuf=[\n"
        repeat
            cpy.b [pbuf+index], c
            if c != 0
                tracef "%c", c
            endif
            cpy (index+1) & (SPACE_SIZE-1), index
        until index == limit
        tracef "]\n"
        cpy index, [pspace+CBUF_INDEX]
        ret
        defend 0
    toolend
```

### 21.2.1 Find the data associated with a cache, or allocate it if necessary - sys/kn/statics/statics_get

This tool finds data associated with a cache, using a pointer to the cache, and a key value. The specified cache must be a -1 initialised cache area, typically in a tool's writable data. The key value should normally be the code address of this tool.

One pointer passed to the tool either points to the simple statics configuration block, or contains 0. If it contains 0 then the data area already exists, and will not be created. The simple statics configuration block must be present until the static area is deleted. This may be ensured by putting it in the data area of the calling tool:

| Field Type | Field Name | Description |
|---|---|---|
| Pointer | WDA_SETUP_TOOL | Tool pointer of caller for tool reference |
| 32-bit integer | WDA_SETUP_LENGTH | Size of user data allocated |

| Pointer | WDA_SETUP_NEW | Initialisation with statics lock on |
|---|---|---|
| Pointer | WDA_SETUP_INIT | Initialisation with statics lock off |
| Pointer | WDA_SETUP_WAIT | Called if data area created but *init* not done |
| Pointer | WDA_SETUP_DEINIT | Deinitialisation routine |
| Pointer | WDA_SETUP_DELETE | Delete areas |

The statics lock is a per process lock used by *sys/kn/statics/statics_get*. It is locked when the *alloc* or *new* routine is called (see below). It is non-recursive, and thus no operations which use the statics facilities are allowed within the process while it is held.

The fields within the block take the following forms:

- WDA_SETUP_TOOL - pointer to tool code for tool containing the cache in its writable data.
- WDA_SETUP_LENGTH - the length of the static area, or -1 (see below).
- WDA_SETUP_NEW - pointer to a new routine, or 0. If WDA_SETUP_LENGTH is -1, then this is instead a pointer to an *alloc* routine.
- WDA_SETUP_INIT: pointer to an *init* routine, or 0.
- WDA_SETUP_WAIT: pointer to a *wait* routine, or 0.
- WDA_SETUP_DEINIT: pointer to a *deinit* routine, or 0.
- WDA_SETUP_DELETE: pointer to a *delete* routine, or 0.

The user initialisation data is passed back to the *alloc/new* and *init* routines if an area is created, and to the *wait* routine if it is called. As a default, if all the procedure pointers are zero, a static area of size WDA_SETUP_LENGTH is allocated and zero initialised. This static area will be automatically removed when the process ends.

### 21.2.2 Delete the statics associated with the key and clear the cache - sys/kn/statics/statics_delete

The static area associated with the cache in the current process is deleted. Static areas are deleted in any case when a process ends, but this tool provides a way of forcibly deleting an area before this point.

It is assumed that the user will have performed any necessary deinitialisation, but the *delete* routine is called if it was specified in the setup data. The tool referenced in the setup data has its reference count decremented.

### 21.2.3 Allocates a static area (used when WDA_SETUP_LENGTH is -1)- alloc

This routine must be provided to allocate a static area if WDA_SETUP_LENGTH is -1. It must also allocate the associated static control area of size WDA_STATICS_SIZE. Both areas may be allocated within the same block.

The routine is called when a request is made for a static area for a particular cache and process where the area does not yet exist.

When using this routine, a *delete* routine must also be provided to remove the data.

This routine is called with the statics lock held, and therefore should only be used to perform very simple tasks to avoid blocking other threads and processes accessing unrelated static areas. More complex initialisation should be performed in the *init* routine. If there several threads within a process may be sharing the static area, such that there could be a race condition as two threads call *sys/kn/statics/statics_get*, then a lock should be used as follows:

- In the *alloc* routine, create and acquire a lock in the static area itself
- Provide a wait routine to wait for the lock to be released
- In the *init* routine, complete the initialisation and then release the lock
- Free the lock in the *deinit* routine

### 21.2.4 Performs initialisation on an automatically allocated static area- new

If setup length is greater than or equal to 0, then the space is automatically allocated. In such circumstances, where WDA_SETUP_LENGTH is not -1, the *new* routine may be provided to perform fast initialisation of this automatically allocated static area. If the *new* routine is not supplied, then the area is simply zero initialised. The space allocated will be automatically freed at the end of the process.

This routine takes a pointer to the newly allocated static area, which has not been initialised to 0 or any other value. The routine returns a statics area pointer which is typically the same as that passed in, but may be changed so that a different address is cached and returned by *sys/kn/statics/statics_get*.

This routine is called with the statics lock held, and therefore should only be used to perform very simple tasks to avoid blocking other threads and processes accessing unrelated static areas. More complex initialisation should be performed in the *init* routine. If there several threads within a process may be sharing the static area, such that there could be a race condition as two threads call *sys/kn/statics/statics_get*, then a lock should be used as follows:

- In the *new* routine, create and acquire a lock in the static area itself
- Provide a wait routine to wait for the lock to be released
- In the *init* routine, do the rest of the initialisation and then release the lock
- Free the lock in the deinit routine

### 21.2.5 If setup length is 0 or more then the space is automatically allocated - init

This routine initialises a statics area where the task is likely to take too long using *alloc* or *new*. It is passed a pointer to the user data area, a pointer to the user initialisation data, and a pointer to a statics control of a specified size.

This routine is called with the statics lock off, so that calls may be made to other routines that use statics.

If there may be a race to create the statics then the *new* or *alloc* routine should set up a lock which *init* can then unlock. The *wait* interface must also be defined in this case.

### 21.2.6 Wait for statics area to be initialised - wait

This routine is called if *init* has not completed and *sys/kn/statics/statics_get* is being called from a different thread within the same process to ask for the same static area. It is necessary that this second thread wait for the first thread's *init* call to complete, and the *wait* routine is called to ensure this. Calls to *sys/kn/statics/statics_get* in the same thread as that in which the statics are being initialised will be honoured without calling wait.

This routine is called with the statics lock off. Calls may be made to other routines that use statics.

This routine is passed a user initialisation data pointer, which is identical to that passed to the current call of *sys/kn/statics/statics_get* rather than the one initialising the area.

### 21.2.7 Deinitialise the statics area - deinit

This routine deinitialises the statics area. This call is made before any statics are automatically freed. The user should put any structured closedown code here. Calls to this interface take place in the reverse order to that used when the areas were set up.

### 21.2.8 Deletes the statics area by freeing it - delete

This routine deletes the static area and frees the memory it occupies. If WDA_SETUP_LENGTH is -1, then this routine must be provided to free a static area and its associated control area, as allocated by *alloc*. If WDA_SETUP_LENGTH is not -1 the static area will be freed automatically, and the user must not free it.

At this point only simple actions for freeing areas should be performed, since other statics may have been freed leaving the environment in a less usable state.

# 22. Kernel Entropy Collector

Numerous applications have requirements for a source of high-quality randomness. In particular, many of the embedded systems upon which int**e**nt is designed to run, such as mobile phones and PDAs, require encryption mechanisms for security purposes.

Most encryption algorithms demand a source of random data, for such processes as 'key generation' and 'secret splitting.' (Encryption of a plaintext message will often be performed by an algorithm in conjunction with a randomly generated key. In some cases the private decryption key will be subjected to "secret splitting" which divides it into discrete shares, each of which is held by a different trustee.)

Most computers that include 'random number generators' in fact make use of pseudorandom number generators (PRNGs). These can be used to create a sequence of numbers that are evenly distributed over a specific range of values, with little correlation between successive numbers. However, the numbers generated by simple PRNG algorithms are not truly 'random,' and thus cannot offer the unpredictability demanded by most encryption processes. In addition, even the cryptographically secure PRNGs require access to a source of entropy with which to 'seed' their own processes.

By their very nature, software mechanisms for the generation of 'random' numbers are deterministic, and therefore predictable. For this reason, they are unsuitable as sources of entropy for encryption algorithms.

The int**e**nt kernel entropy collector, on a small scale, adopts the principle upon which much military-grade cryptographic hardware has been based. Historically, such machines have often included hardware specially designed to generate entropy. In fact, such specialised hardware is unnecessary. The ordinary hardware associated with many PCs can be used to generate entropy at a reasonable rate.

It is such ordinary sources of entropy that the int**e**nt entropy collector is designed to exploit. The disorder in a system is measured and 'collected' through examination of random events. The latency of a hard disk, for example, can be used in this way, since the time the disk takes to respond to requests is influenced by the air turbulence around the spinning platter. The time lapse between user key presses or network events, the noise from a semiconductor device and the least significant bits of an audio input are also random factors that can be monitored.

Since this source of randomness is not based upon a software algorithm, the int**e**nt kernel entropy collector is able to provide a means of accumulating real random data, collected from device drivers. If a large quantity of random data is required, then the entropy collected can be used as a reliable seed for a cryptographically secure PRNG.

## 22.1 Entropy Collection Overview

Since this form of entropy originates in the hardware, it is not feasible for individual applications to collect the entropy they require directly.

Instead, each device driver is made responsible for tapping the entropy generated by the device with which it is associated. The entropy collected by all drivers is collected by the kernel entropy collector, which makes it available to all applications that require it.

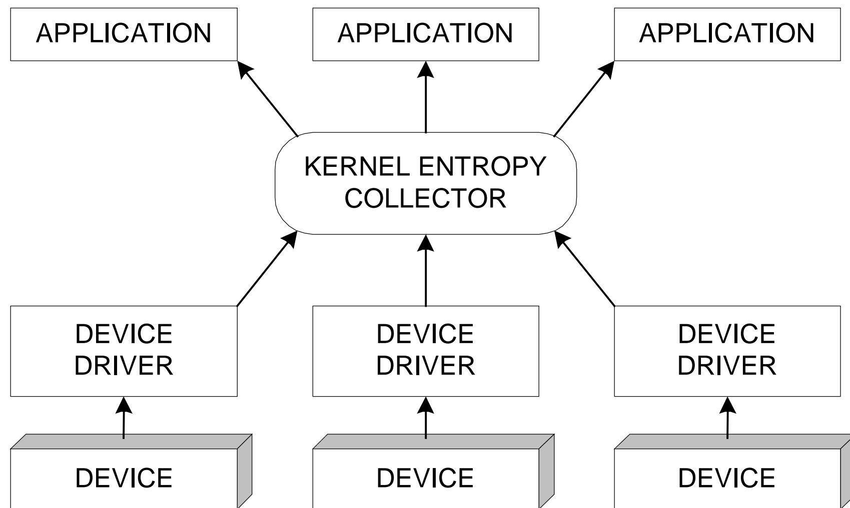


**Diagram 10 - The Kernel Entropy Collector**

The random data supplied by the device drivers is transferred into a data pool maintained by the kernel entropy collector. This collector maintains a count of the current number of entropic bits contained within this pool. One method is employed to ensure that additions to the entropy in the pool are paralleled by a comparable increase in this count. Another method ensures that when entropy is extracted from the pool, the count decreases to the appropriate degree.

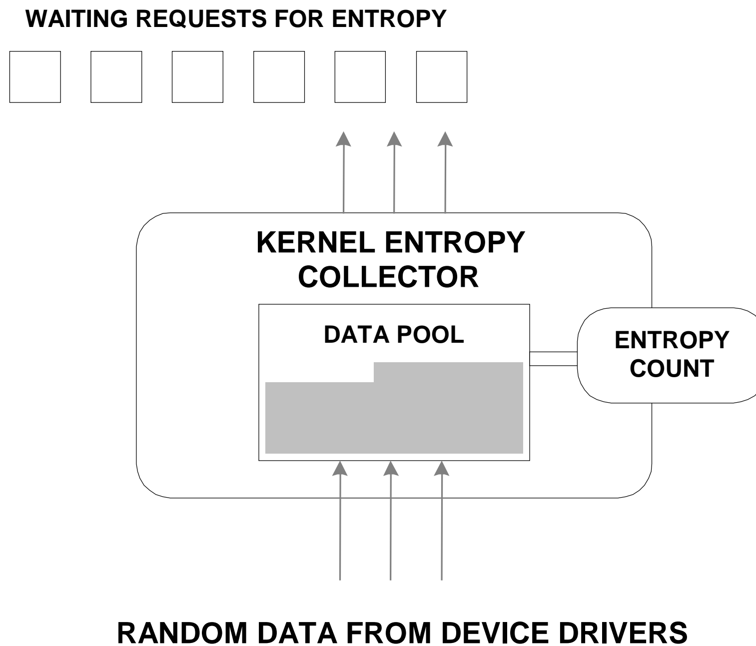


**Diagram 11 - The Kernel Entropy Collector Data Pool**

Not all embedded systems will require this entropy collection facility. Some systems may have no security requirements, and hence no need for a source of high-quality randomness. Others may provide no source of randomness, and thus will be unable to support this facility.

In such cases, a null version of the kernel entropy collector will be used, instead of the full, functional version. At sysgen time it is possible to specify whether the system will use the real entropy collector, or a group of stub routines which only provide the appropriate API for device drivers.

If the real entropy collector is to be used, the sysgen steering file must contain the following directive:

```
#include sys/kn/entropy/entropy.sys
```

If this is not included, then the calls made to the collector in order to add entropy to its pool become no-ops, and entropy cannot be extracted.

## 22.2 The Kernel Entropy Collector

In many cases, device drivers generate random data during interrupt service routines. For this reason, entropy collection has been designed to cause as little delay as possible, and to avoid table lookups and lengthy list searches.

In order for an addition to be made to the random data in the entropy collector's pool, a simple *qcall* is made to the collector. Like most other facets of the kernel, the entropy collector itself is a collection of tools. The references required by these tools have been statically fixed up, and this data stored in a named data area.

The random data collected by the kernel is mixed using SHA-1, a cryptographic one-way hash function. After mixing, the data contains maximum entropy. Thus, if the hash function is given an N-bit input containing M bits of entropy, and yields an N-bit output, any M bits of the output will contain M bits of entropy. In effect, a sufficiently large subset of a hashed bit sequence will contain, in the absence of the rest of the sequence, the same amount of entropy as that in the original bit sequence.

Once the pool contains mixed data, it would be possible to copy portions of it immediately, and thus generate pure entropy. However, this would be dangerous, since it would make it easier for malicious parties to generate inputs partially correlated with the current pool. Therefore, before the data is used by an application, it is operated upon by another hash function.

### 22.2.1 Entropy Collector Input

One result of using physical sources of randomness is that each entropy generator produces its random data in a different form. The kernel entropy collector might, for example, be receiving data of this sort from a driver monitoring the latency of the hard disk. Although the latency of the disk might have a minimum bound of 5 milliseconds and maximum bound of 100 milliseconds, most values would be distributed towards the lower end of this range. At the same time, other sources might be producing random data in which all values were spread through a different range, or in a different fashion.

The entropy collector is designed to accept random data in any form, regardless of its density or distribution. The individual callers do not need to condense the random data into pure entropic bits.

It is not even necessary for the random data to be 'unbiased' by the caller. (Unbiasing a sequence of values entails spreading the values over a specified range, different from that which they had originally spanned.)

### 22.2.2 Entropy Collector Output

The entropy collector is designed so that at no point can it output more data than there is entropy remaining in the data pool. This ensures that successive outputs are in no way correlated.

Data emitted by the collector should have maximum entropy, i.e. one bit of entropy per data bit. Thus applications do not need to 'fold' the data. A bit sequence that is 'folded' or 'compressed' is transformed into a shorter bit sequence, while retaining the same amount of entropy.

In order to avoid a waste of entropy, each application is able to request a precise number of bits of entropy. These are not restricted to any alignment.

## 22.3 Kernel Entropy Collector Functions

### 22.3.1 Add entropy to the kernel collector pool - *sys/kn/entropy/add*

This function adds random data to the entropy pool. This data need not be unbiased. Parts of the data may be non-random, correlated or even selected maliciously without causing problems, providing that they are in no way correlated to the contents of the pool.

The entropy input is an estimate of the amount of entropy contained in the random data. If this value given is an underestimate, this causes no problems, since the system will 'play safe', assuming that the data is less random than it is. However, the value in this parameter must never be an overestimate, since this might lead non-random data to be treated as random.

Data that has been passed to *sys/kn/entropy/add* loses all entropy, from the point of view of the entropy collector. If inputs are correlated, the total of their claimed entropies must be no higher than the true total entropy of all the inputs.

### 22.3.2 Add timer entropy to the pool - *sys/kn/entropy/add_time*

This function adds the current time to the entropy pool, as random data. The actual entropy added by this is estimated through use of an uncertainty parameter, U.

The semantic of the parameter U is as follows. The caller guarantees that observation of the system and the data stored in the entropy pool cannot be used to predict the time at which the function is called with an uncertainty of less than U nanoseconds.

For example, outside observation might reasonably establish the time of a keypress might to within 0.1 seconds (10^8 nanoseconds), through outside observation. If a call is to be made at the time of this keypress, then the time of its occurrence can also be predicted to within 10^8 nanoseconds. U should, therefore, be given the value 10^8, since this represents the level of uncertainty in nanoseconds.

When calculating an appropriate value for U, an underestimate is safer than an overestimate. The latter might result in the entropy collector attributing randomness to non-random data, and ultimately supplying applications with data that is not truly random.

### 22.3.3 Register the timer clock resolution - *sys/kn/entropy/reg_time*

This tool is used to register the resolution of the system clock. This is necessary for estimation of the entropy in times of random events, that is to say, the margin of unpredictability caused by system disorder. This estimation is required in order to calculate the uncertainty parameter for *sys/kn/entropy/add_time* (see above).

The *sys/kn/entropy/reg_time* function should only ever be called by the timer device while it is initialising.

The semantic of U, the uncertainty parameter for this function, is as follows. The timer device guarantees that, when it updates the system time, the differences between two time values will be correct with an uncertainty not greater than U nanoseconds. A reliable 1024Hz timer, for example, would set U to 976563 or greater. When calculating the uncertainty parameter, it is safer to overestimate than to underestimate.

It should be noted the uncertainty parameter is equivalent to the precision of the time source. Distinctions must be drawn between the "precision," the "resolution" and the "accuracy" of a timer. The "resolution" is the difference between consecutive representable times. The "precision" is the difference between successive values of the clock. Accuracy is the maximum absolute difference between the time shown on the clock and the actual current time.

The intent clock, for example, has resolution of one nanosecond. Its precision is platform dependent, but is 0.1 seconds on Linux and 5 milliseconds on the Shboom board. On intent systems, the timer will usually have an accuracy of a few seconds.

### 22.3.4 Extract entropy from collector pool - *sys/kn/entropy/get*

This tool is called by an application to extract entropy from the entropy pool. If the amount of entropy in the pool is inadequate to satisfy the request made by this function, the system will wait for more entropy to be collected. Thus the time taken for *sys/kn/entropy/get* to return the required entropy is of arbitrary length. The collector will never return more entropy than is available in its data pool.

The two integer inputs specify the bounds of the range of quantities of entropy that are acceptable. In most cases the application requires an exact number of bits, and consequently both these values will be the same. If the values are different, the collector will return as soon the available entropy equals or exceeds the minimum requirement. The collector will return as much of the available entropy as is possible without exceeding the specified maximum.

This function returns unbiased random data, uncorrelated to any data generated by a different call. It is computationally infeasible to determine the state of the entropy pool from this data. After a successful return, it is equally infeasible to determine the data returned from the state of the entropy pool.

The returned data is stored in provided buffer, starting at the location indicated by the input pointer, and filling consecutive bytes. If the number of bits returned is not a multiple of 8, the excess bits are stored in the low-order bits of the appropriate byte. The high-order bits of the byte are cleared. Further bytes, beyond those required to store the result, are unaffected.

Some entropy may be successfully returned even if the operation as a whole fails, but should be treated with caution.

### 22.3.5 Extract entropy from collector pool (alternative interface) - sys/kn/entropy/getrand

Like *sys/kn/entropy/get*, this function provides synchronous entropy generation, extracting entropy from the collector data pool. The *sys/kn/entropy/get* function offers the primary interface, but *sys/kn/entropy/getrand* uses a different interface, one more suited to cases where the entropy collector must be interchangeable with other bit stream sources.

### 22.3.6 Extract entropy asynchronously - *sys/kn/entropy/get_async*

This tool queues an asynchronous request for entropy. The request is processed in the same way as that issued by *sys/kn/entropy/get*, but instead of waiting for the correct amount of entropy to be made available, this function returns immediately. Its status return gives no information as to whether the requested entropy has been extracted. Instead it only indicates whether the request was successfully queued.

When the request has been processed, the callback function specified in the control structure is queued (see the section on "Callback Manipulation Functions"). The request can be terminated early by using *sys/kn/entropy/get_abort*.

The control structure, of size ENTROPY_SZ, is defined in the section on "Data Structure Definitions."

This structure begins with a CALLBACK structure, which must be filled in by the caller to specify which function needs to be called back. The caller also needs to fill in the following:

`pointer ENTROPY_BUFFER` - The output buffer
`int32 ENTROPY_MIN` - The minimum entropy to return (in bits)
`int32 ENTROPY_MAX` - The maximum entropy to return (in bits)

When the callback is called, the following fields will have been filled in:

`int32 ENTROPY_STATUS` - Status (0 if success, error code if error)
`int32 ENTROPY_RET` - The entropy actually returned (in bits)

### 22.3.7 Cancel asynchronous entropy request - *sys/kn/entropy/get_abort*

This function aborts the asynchronous entropy extraction request initiated by *sys/kn/entropy/get_async*, and associated with the specified control structure. This can be called at any time after the request has been queued and before the callback function is called.

If this tool is called after the callback is queued, then the call has no effect. If the callback has not yet been queued, it will be so when this tool returns. No callbacks will be taken while the call is in progress.

When the callback function is finally called, the ENTROPY structure will be updated to indicate the actual status of the request. If it has completed successfully, this is indicated. A request that was actually interrupted before completion will have a status of ECANCELED.

# 23. Kernel Time functions

### 23.1.1 Get the kernel time - sys/kn/time/get

This function returns the current kernel time, in nanoseconds since the intent system was booted.

# 24. Data Structure Definitions

## 24.1 Process Control Block Structure

The process control block (pcb) structure contains the scheduling parameters of a particular process, and is defined as shown below.

| Field Type | Field Name | Description |
|---|---|---|
| 32-bit integer | PCB_PRIORITY | Process priority |
| 32-bit integer | PCB_STATE | Process state |
| 64-bit integer | PCB_BCET | Best-case execution time |
| 64-bit integer | PCB_WCET | Worst-case execution time |
| 64-bit integer | PCB_PERIOD | Period (for periodic processes) |
| 64-bit integer | PCB_DEADLINE | Deadline |
| 32-bit integer | PCB_EXITVAL | Exit status (only valid when process has run and exited) |
| 32-bit integer | PCB_PPID | Parent process ID |

The overall size of the data structure is PCB_SIZE.

Contents of fields:

- PCB_PRIORITY contains the base priority of the process. In a priority scheduler, this is the only parameter which affects scheduling decisions. The highest priority process in the READY state is always scheduled.

- PCB_STATE contains the process state. This is a combination of the following values, each of which is represented by a single bit: PSTAT_DORMANT, PSTAT_RUN, PSTAT_READY, PSTAT_SUSPEND, PSTAT_SLEEP. Normally only one of these bits is set. The only valid combination is PSTAT_SLEEP + PSTAT_SUSPEND, which indicates that the process is in the SLEEP+SUSPEND state.

- PCB_WCET contains the worst case execution time of the process. This may be used by a dynamic scheduler (Earliest Deadline First, Minimum Laxity First, Maximum Urgency First, etc.) to calculate the dynamic priority of the process.

- PCB_BCET contains the best case execution time for the process. This may be used as part of a deadline violation detection mechanism.

- PCB_PERIOD contains the time between triggering events for a periodic process. For a non-periodic process, period should specify the worst case interarrival time (the minimum possible period). This may be used for scheduling decisions in a dynamic scheduler.

- PCB_DEADLINE contains the process's deadline time. The operating system may use this entry to detect deadline violation. In some cases, it may be possible to detect the deadline failure before the process is scheduled (e.g. if the deadline cannot be met due to pre-emption by a higher priority task). For periodic processes, the deadline may often be found by adding the period to the time at which the process was triggered.

- PCB_EXITVAL contains the contents of the PROC_EXITVAL field in the process's global data area. When a process is in the DORMANT state, having completed its execution, this field usually

contains the exit status of the process. If the process died due to an uncaught signal, this field contains the signal number which caused the process to die.

- PCB_PPID contains the process ID of the parent of the process whose parameters are under examination.

## 24.2 Spawn Structure

The spawn data structure specifies the parameters for process creation.

The structure starts with a 32-bit integer containing the size of the entire structure, followed by a series of records.

The first word in each record is a 32-bit integer whose most significant 8 bits contain the record type, and whose least significant 24 bits contain the record length (the offset from the start of the current record to the start of the next). The contents of the body of each record depends on the record type.

The spawn structure is terminated by a record whose type/size field with a value of 0.

The table below shows the record types which are currently defined. The 'Offchip' column indicates whether each record type can be used in spawn-structures being sent offchip (ie, to be run on another processor). If a record type has an 'Offchip' value of 'no,' this is usually because the record type involves transmitting a pointer value, which is useless on the target processor.

The following types are currently defined:

| **Data type** | **Type Name** | **Description** | **Offchip** |
|---|---|---|---|
| No data | SPAWN_END | Terminal record | Yes |
| Null-terminated string | SPAWN_NAME | Name of main program tool | Yes |
| 32-bit integer *N*, sequence of *N* null-terminated strings | SPAWN_ARGS | Program argument. The integer indicates how many strings there are in the sequence. The strings specify the arguments passed to the process. | Yes |
| Sequence of two null-terminated strings | SPAWN_ENV | Environment string pair to be declared as local variables for the program (name, value) | Yes |
| 32-bit integer, 32-bit pointer, 32-bit pointer, 32-bit integer | SPAWN_FD | File descriptor template, in the format fd number, device handle, device instance pointer, processor number. (Note: the programmer is discouraged from making alterations to SPAWN_FD.) | Yes |
| Null-terminated string | SPAWN_STACKNAME | Name of memory object for stack allocation | Yes |
| Null-terminated string | SPAWN_DATANAME | Name of memory object for data memory allocation | Yes |
| 32-bit pointer | SPAWN_STACKOBJ | Address of memory object for stack allocation | No |
| 32-bit pointer | SPAWN_DATAOBJ | Address of memory object for data memory allocation | No |
| Sequence of bytes | SPAWN_GLOBALS | Initialised global data area for child (size if given by type/size field for record) | Yes |
| 32-bit pointer | SPAWN_STATICS | Address of statics object to give to child | No |

| | | | |
|---|---|---|---|
| Null-terminated string | SPAWN_STDIN | Textual name of file/device to open as process's stdin | Yes |
| Null-terminated string | SPAWN_STDOUT | Textual name of file/device to open as process's stdout | Yes |
| Null-terminated string | SPAWN_STDERR | Textual name of file/device to open as process's stderr | Yes |
| 32-bit pointer | SPAWN_SIGNAL | Signal table object pointer to give to child | No |
| 32-bit pointer | SPAWN_FILETAB | File table object pointer to give to child | No |
| 32-bit pointer | SPAWN_ENVOBJ | Environment object pointer to give to child | No |
| No data | SPAWN_PARENT | Parent process ID from PCB_ structure (presence of record is a boolean flag) | Yes |
| No data | SPAWN_LOCALPID | Flag: Allocate the process ID for child process from local PID pool (presence of record is a boolean flag) | Yes |
| 32-bit pointer | SPAWN_LOADER | Tool loader to be used when loading the main tool for the child process | No |
| 32-bit integer | SPAWN_STACKSIZE | Size of the stack | Yes |
| 32-bit unsigned integer | SPAWN_SIGMASK | Signal mask to be used by process | Yes |
| 32-bit integer | SPAWN_STKLIMIT | Limit to the size of the stack | Yes |

It should be noted that spawn structures may or may not contain padding to align the records to 4-byte boundaries. The programmer cannot assume the records are aligned, and any word accesses into the structure must use the non-aligned VP instructions.

## 24.3 Event Flag Information Structure

The event flag information data structure is defined as follows:

| Field Type | Field Name | Description |
|---|---|---|
| Unsigned 32-bit integer | EVFI_PATTERN | Event flag pattern |
| 32-bit integer | EVFI_TASK | Details of waiting processes |

The overall size of the data structure is EVFI_SIZE.

Contents of fields:

- The EVFI_PATTERN field contains the event flag pattern at the time that the structure was filled in (usually by the *evf_info* function).

- The EVFI_TASK field contains 0 if there are no processes waiting on the event flag, or a non-zero value if there are one or more processes waiting.

## 24.4 Mail Message Header

The structure of mail message headers is defined as follows:

| Field Type | Field Name | Description |
|---|---|---|
| intent list node | MSG_LISTNODE | List node for use by the kernel |
| 64-bit integer | MSG_SENDER | Mailbox ID of sender |
| 32-bit integer | MSG_LENGTH | Message length |

At offset MSG_LISTNODE , there is a list node which allows the parts of message header structure to be contained within a standard intent list. This is not a field of the structure, but is inherited from the list node structure.

The overall size of the data structure is MSG_DATA (which is of course also the offset at which the message data is found).

Contents of fields:

- The MSG_SENDER field contains the mailbox ID of the sender. This is often used when a reply is required.

- The MSG_LENGTH field contains the length of the message, including both the message header and the message body.

## 24.5 Memory Flush List Nodes

The memory flush list contains nodes with the following structure:

| Field Type | Field Name | Description |
|---|---|---|
| intent memory node | MFL_NODE | List node for use by the kernel |
| Pointer | MFL_HANDLER | Routine to call on flushing |
| Pointer | MFL_MEMOBJ | Memory object from which this memory was allocated |
| 32-bit integer | MFL_PRIORITY | Flushing priority |

At offset MFL_NODE there is a list node, containing information which is used by the kernel. This is not a field of the structure, but is inherited from the list node structure.

The overall size of the data structure is MFL_SIZE.

Contents of fields:

- The MFL_HANDLER field of the structure contains a pointer to the function to be called when the memory flush list is processed.
- The MFL_MEMOBJ field indicates which memory object allocated the memory associated with this node. When a memory object requires to flush memory, only the handling functions of nodes whose memory will be freed back to that object will be called. If the contents of the field is NULL, then all flush operations will call this handler.
- The MFL_PRIORITY field contains the flushing priority of the node, which affects the order in which the handlers are called when flushing occurs. If the first handlers called free sufficient memory, then lower priority handlers might not be called. The priority should be one of: MFP_PREFER, MFP_OFTEN, MFP_NORMAL, MFP_SELDOM or MFP_EMERGENCY.

## 24.6 Callback Data Structure

This data structure is used when posting callbacks to indicate device I/O completion, or other events. The structure is defined as follows.

| Field Type | Field Name | Description |
|---|---|---|
| intent list node | | List node for use by the kernel |
| Pointer | CALLBACK_DATA | Application-specific data to be passed to handler function |

| Pointer | CALLBACK_HANDLER | Address of handler function |
|---|---|---|
| Pointer | CALLBACK_PID | Process context details |

The overall size of the data structure is CALLBACK_SIZE

The list node at the beginning of the data structure is the node used to link the structure onto the process's callback list. This is not a field of the structure, but is inherited from the list node structure.

Contents of fields:

- The CALLBACK_DATA field contains the user-data to be passed to the handler function when a callback occurrs. However, the exact format of the data is application specific, and hence cannot be described here in any detail.
- The CALLBACK_HANDLER field contains a pointer to a callback handler function which must be called and passed the value in the *callback_data* field. Further detail of this function can be found in the chapter on Callbacks.
- The CALLBACK_PID field gives the identifier of the process within whose context the callback function is running.

## 24.7 Notify Data Structure

The notify data structure is a block of memory with the following fields defined:

| **Field Type** | **Field Name** | **Description** |
|---|---|---|
| intent Callback Structure | NOTIFY_CALLBACK_DATA | A callback data structure as defined earlier in this section. |
| Pointer | NOTIFY_EVENT_DATA | Pointer to data associated with the event. |
| Pointer | NOTIFY_EVENT_ID | Event ID (integer) of the event. |

At offset NOTIFY_CALLBACK_DATA, there is a callback data structure. Where an event notification is being handled in the context of the process that has subscribed to be notified, this callback is linked to the process's callback list in the same fashion as any other callback. If the notification is to be handled in the context of the generator of the event, then the callback data structure is not linked to the callback list of the generator. However, it is still used to store the pointer to the handling function, and the subscriber data pointer.

This is not a field of the data structure, but is inherited from the callback structure.

The overall size of the data structure is NOTIFY_SIZE.

Contents of fields:

- The NOTIFY_EVENT_DATA field contains user-defined data associated with the event.

- NOTIFY_EVENT_ID gives an ID code derived from the name of the event, using kernel atom functions.

## 24.8 AVL tree

An AVL tree is a piece of memory of the appropriate size (see below) which has been passed as a parameter to one of the *sys/kn/avl/init** functions.

| | Fixed stack AVL tree | | Relocatable stack AVL tree | |
|---|---|---|---|---|
| | **C** | **VP** | **C** | **VP** |
| Structure size | *sizeof(ELATE_AVL TREE)* | *avl_struct_size* | *sizeof(ELATE_AVL TREE_RS)* | *avl_struct_rs_size* |
| Initialisation functions | *kn_avl_init()*<br>*kn_avl_init_dup()* | *sys/kn/avl/init*<br>*sys/kn/avl/init_dup* | *kn_avl_init_rs()*<br>*kn_avl_init_dup_rs()* | *sys/kn/avl/init_rs*<br>*sys/kn/avl/init_dup_rs* |

Each element within the tree needs a structure of size *avl_item_size* (or *sizeof(ELATE_AVLNODE)* if using C) embedded within it at the same offset from the beginning of the element. This offset need not be 0. Neither the AVL tree header nor the structure embedded within tree elements contains any user fields.

When using the C interfaces, the type ELATE_AVLITEM is available. This may denote a pointer to an element already in the AVL tree. Alternatively it may denote a pointer to an element to be inserted into the tree or compared against elements in the AVL tree. It is not advised to use an (ELATE_AVLNODE *) as a pointer to an AVL tree element, as the ELATE_AVLNODE structure is not necessarily positioned at the beginning of the element.

## 24.9 Timer Structure

A timer is a data structure containing the following fields:

| Field Type | Field Name | Description |
|---|---|---|
| intent memory node | TT_NODE | List node for use by the kernel |
| Pointer | TT_HANDLER | Pointer to function to call when timer expires, or constant specifying other action |
| 64-bit integer | TT_EXPIRE | Expiry time |
| 64-bit integer | TT_PERIOD | Period of timer |
| 32-bit integer | TT_PARAM | Parameter to user-specified timer handler function or other action |
| 32-bit integer | TT_PRI | Process ID of timer ‘owner’ |
| 32-bit integer | TT_MODE | Operation flags |

At offset TT_NODE there is a memory structure which is used by the system. This is not a field of the data structure, but is inherited from the memory node structure.

The overall size of the data structure is TT_SIZE.

Contents of fields:

- The TT_MODE field contains flags specifying the operation of the timer. One of the following flags must be set:

  - TTF_MONO
    This value specifies that the timer is a monoshot timer. After its initial expiry, this timer becomes dormant.
  - TTF_PERIOD
    This value specifies that the timer is a periodic timer. Each time this timer expires, the expiry time is reset to the value in *tt_period*.

  Also, one of the following flags must be set:

- TTF_ABS
  This value specifies that the time given in *tt_expire* is absolute.
- TTF_REL
  This value specifies that the time given in *tt_expire* is relative to the time at which the timer was set.

- The TT_EXPIRE field's contents depends on the type of timer. If the timer is a monoshot timer, the TT_EXPIRE field contains the timer's expiry time in nanoseconds. If the timer is a periodic timer, the TT_EXPIRE field contains the start time of the timer, also in nanoseconds. The expiry time is absolute if the timer is an absolute timer and relative if the timer is a relative timer.

- The TT_PERIOD field's contents depends on the type of timer. If timer is a periodic timer, then the field contains the period of the timer, in nanoseconds, as a relative value. The value gives the expiry time to which the timer is set after each expiry. If the timer is a monoshot timer, the TT_PERIOD field is not used and is set to 0.

- The TT_PRI field contains the priority of the process that owns the timer, and is filled in by the system when the timer is set.

- The meaning of the TT_PARAM field depends on the value of the *tt_handler* field, as described below.

- The TT_HANDLER field contains a pointer to the function to be called when the timer expires, or a constant specifying other action. It contains one of the following values:

  - TT_WAKEUP
    This value specifies that the TT_PARAM field of the timer structure contains the process ID of a process whose timeout has expired. The *sys/kn/proc/wake* function should be called on the process.
  - TT_DEADLINE
    This value specifies that the TT_PARAM field of the timer structure contains the process ID of a process which has failed to meet its deadline. A deadline exception should be generated for the process.
  - OTHER

Any other value is interpreted as the address of a timer handler to be called when the timer expires. The TT_PARAM field is passed as a parameter to the handler.

## 24.10 Entropy Collector Control Structure

This structure contains details of an asynchronous request for entropy made to the kernel entropy collector.

| **Field Type** | **Field Name** | **Description** |
|---|---|---|
| intent callback structure | | Callback structure |
| 32-bit integer | ENTROPY_CBSET | Callback check |
| Pointer | ENTROPY_BUFFER | Buffer into which results are written |
| 32-bit integer | ENTROPY_MIN | Minimum entropy to return |
| 32-bit integer | ENTROPY_MAX | Maximum entropy to retun |
| 32-bit integer | ENTROPY_STATUS | Status |
| 32-bit integer | ENTROPY_RET | Entropy returned |

At the beginning of the data structure there is an inherited callback structure; this is not a field of the data structure. The callback is queued when the request is processed.

The overall size of the data structure is ENTROPY_SZ.

Contents of fields:

- The ENTROPY_CBSET field indicates whether or not the callback has been set.
- The ENTROPY_BUFFER field contains a pointer to a buffer in to which the requested random data should be written.
- The ENTROPY_MIN field contains the minimum number of bits that could be acceptably returned in response to the request.
- The ENTROPY_MAX field contains the maximum number of bits that could be acceptably returned in response to the request.
- The ENTROPY_STATUS field indicates the status of the request, and whether it has succeeded (in which case it contains 0), or been prevented by an error (when it will contain the relevant error code).
- The ENTROPY_RET field contains the number of bits of random data that have been returned, in the case of the request being met.

# 25. Glossary

**Deadline**
The deadline of a task is the latest time by which the task must have completed in order to be considered successful.

**Deadline Monotonic Scheduling.**
This is a static analysis method in which task priorities for a real time system are assigned solely on the basis of the deadline of the tasks. The task with the shortest deadline is given the highest priority.

**Deadlock**
The condition in which every response in a set of responses is using a resource and is also waiting for another resource which must first be relinquished by another response in the set. In effect, they are all waiting for each other.

**EDF**
Earliest Deadline First. Scheduling method where the tasks are ordered on the dispatch list according to their deadlines. The task whose deadline is closest is dispatched first.

**Highest Locker Protocol**
This is a synchronisation protocol, in which the user of a data object executes at a priority which is immediately greater than that of the highest priority task capable of accessing the data object.

**Laxity**
The laxity is the difference between the current time and the start deadline.

**MLF**
Minimum Laxity First. A scheduling method in which the tasks are ordered based on their laxity. The task with the lowest laxity value is dispatched first.

**MUF**
Maximum Urgency First. A modification of the MLF scheduling method, in which tasks are categorised into several priority levels. Tasks with higher priorities are always dispatched first. Within a priority level, the task with the minimum laxity is dispatched first.

This modification makes it possible to control which tasks fail their deadlines in a transient system overload (in EDF and MLF it is not possible to control which tasks fail).

**Priority Inversion**
This is a condition that occurs when a high priority process is blocked waiting for a resource (such as a mutex) which is owned by a low priority process.

Since the low priority process is in a critical section, it would normally execute quickly and release the lock, allowing the higher priority process to get the lock and continue. However, the low priority process may be pre-empted by a medium priority process, which is not in a critical section and may therefore fail to finish executing for an arbitrarily long time. This could cause the high priority process to fail to meet its deadlines. This is called unbounded priority inversion.

**PCP**
Priority Ceiling Protocol. This is a method for preventing priority inversion. Each resource has a priority ceiling assigned to it, as in the highest locker protocol. An operating system variable is maintained, which is the highest priority ceiling of all locked resources. This variable is called the current system ceiling.

Unless a task holds the resource which set the current system ceiling, that task can only lock another resource if that resource's priority ceiling is higher than the current system ceiling.

If a task is prevented from locking a resource, the task holding the resource inherits the priority of the blocked task.

**Priority Inheritance**
This is a method for preventing priority inversion. It means that when a process owns a resource, it executes at the higher of its priority and the highest priority of any process blocked waiting for the resource. Thus, if a high priority process pre-empts a low priority process and attempts to get a resource which is owned by the low priority process, the low priority process 'inherits' the priority of the high priority process, and runs at high priority until it releases the resource.

**Priority Signalling**
This is a method for preventing priority inversion. In this method, when a process attempts to get a resource that is locked by another (lower priority) process, the process which owns the lock receives a signal SIGRES or an exception. The signalled process should immediately clear up its data structures and unlock the resource, allowing the higher priority process to successfully acquire the lock.

This method provides potentially lower (and more predictable) latency for high priority processes where the lock is held by a lower priority process for a significant amount of time, but it has higher overhead than the priority inheritance or priority ceiling methods. This method causes low priority processes to fail to complete critical sections if a higher priority process requires the resource. The low priority process usually retries the critical section until it succeeds.

**RMS**
Rate Monotonic Scheduling. This is a static analysis method where task priorities for a realtime system are assigned based solely on the period of the tasks. The task with the shortest period (the highest frequency) has the highest priority.

Since this method is completely calculated at design time, and is fully static at run-time, the scheduler is very simple. It need only dispatch the highest priority task that is in the READY state.

This method provides mathematical guarantees of task-set schedulability for periodic tasks. However, the CPU utilisation must be significantly lower than 100% for this guarantee to hold. For a particular set of tasks with known worst case execution times, etc, the maximum possible CPU utilisation for guaranteed schedulability can be calculated. If this threshold is exceeded, some tasks may fail to meet their deadlines. The order in which tasks will fail to meet their deadlines is always lowest priority first.

There are numerous extensions to the method to provide for aperiodic tasks and dynamic priority modification.

**Start Deadline**
The start deadline is the latest time at which the task must have started in order to be considered successful.

**Time to Deadline**
The time to deadline is the amount of time between the current time and the deadline.

*© Tao Group Ltd or Tao Systems Ltd. 2000, 2001. All Rights Reserved.*

*Copyright in the software either belongs to Tao Group Ltd or Tao Systems Ltd. The software may not be used, sold, licensed, transferred, copied or reproduced in whole or in part or in any manner or form other than in accordance with the licence agreement provided with the software or otherwise without the prior written consent of either Tao Group Ltd or Tao Systems Ltd.*

*No part of this publication may be reproduced in any material form (including photocopying or storing it in any medium by electronic means and whether or not transiently or incidentally to some other use of this publication) without the written permission of the copyright owner.*

*Elate®, int**e**nt® and the Tao logo are registered trademarks of Tao Group Ltd.*
*Digital Heaven™ is a trademark of Tao Group Ltd.*
*The rights of third party trademark owners are acknowledged.*