System Programming Guide

Version 1.27

1. Introduction

This document aims to provide the system programmer with an overview of the int**e**nt® and Elate® architecture. The guide intends to introduce the programmer to key concepts and provide a cookbook of initial ideas. The programmer can then build on this and improve his knowledge of int**e**nt by consulting the appropriate reference manual for further in-depth information.

This guide assumes that the programmer has worked through the *'VP Tool Programming Guide'* and the '*Object Based Programming Guide'* and is familiar with the basics of assembling and running VP tools. The topics covered in this guide include:

- Kernel
- Translators
- Platform Isolation Interface (PII)
- Device drivers

2. System Portability

The term portability is usually used to refer to the porting of the applications that use an operating system to new platforms or processors. However, as a result of its unique structure and highly efficient translation procedure, Tao's software may be ported in its entirety, libraries and all. Rather than catering to the peculiarities of a particular processor, Tao has designed its technology with portability in mind. The operating system is written as if to run upon a 'virtual processor,' an imaginary, 32-bit, RISC (Reduced Instruction Set Computer) little-endian machine. Depending upon whether porting is to a new processor, a new platform, or both, different interface elements will need to be written.

To port to a new processor, it is necessary only to write the Translator, the CPU Isolation Interface (CII) and sometimes certain processor specific libraries. Since porting to a new processor tends to involve interfacing the system with a new platform, in many cases the requirements of a platform port will also be necessary. Porting to a new platform requires a re-implementation of the Platform Isolation Layer (PIL). This is a convenient group term for all the code written to fit int**e**nt to a specific platform. This layer is comprised of the Platform Isolation Interface (PII) and the Platform Specific Device Drivers.

As such, whereas normally an operating system is defined as being designed to control the hardware of a specific data-processing system to allow application programs to run upon it, Tao's technology draws a distinction between the portable and non-portable system components. The portable components, including Tao's Java™ implementation and media libraries, are generally referred to as being part of the binary portable int**e**nt platform, whereas all non-portable components, such as translators and device drivers, are referred to as being part of the Elate operating system. Certain components may be present for either int**e**nt or Elate.

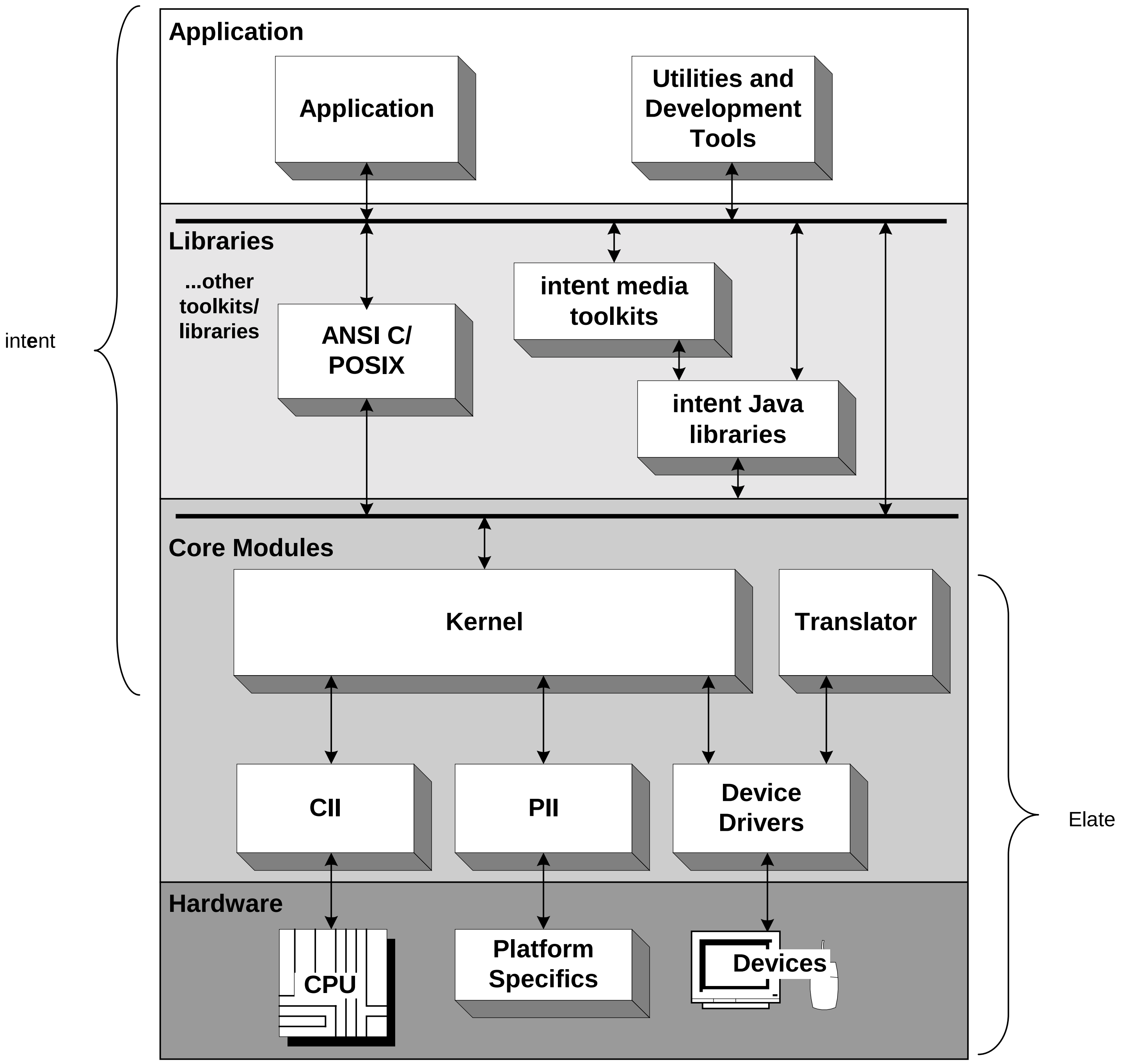

Figure 1. System Overview.

This document details the main components of both the Elate and int**e**nt systems. For these purposes, intent consists of the following:

♦	ANSI	- C/POSIX libraries
♦	Java	- PersonalJava 1.1.3 / 1.2 Libraries
♦	Media	- Media libraries
♦	Kernel	- Realtime Kernel
♦	Sysgen/Sysbuild	- System image builder
♦	DFA	- Data flow analyser
♦	Ebug/Fbug	- Win32 and Linux hosted debugger

♦ Assemblers	- VP and native
♦ Compilers	- C and C++ (GNU ports)
♦ Editors	- JOVE (a simple Emacs clone)

Conversely, Elate consists of the following:

- Platform Isolation Interface (PII)
- Device Drivers
- CPU Isolation Interface (CII)
- Translators

3. Processor Ports

3.1 The Translator

When effecting a processor port, the only substantial piece of code that needs to be written is the translator. Each translator is specific to a different processor. A translator may be created for any type of processor, whether a Digital Signal Processor, a Microprocessor, or a Microcontroller.

Using the same source code, translators designed for different processors of the same family will often generate slightly different code. This occurs because one contains instructions the other lacks, or because optimisation considerations differ between these processors. Compiling or hand coding for a generic family cannot offer this degree of efficiency.

3.1.1 Translation and VP Code

Applications are stored as collections of tools. Tools are objects comprised of executable code. A 'thread' is a separately schedulable entity. A process is a group of one or more threads sharing a common data area but with each thread having its own stack. An application is a structure built from one or more processes.

Two different mechanisms are available for loading tools into memory. Both techniques are designed to ensure that tools exist in memory only when required. In both cases, translation takes place during load time.

Sysgen

Tao provides a utility called 'Sysgen,' the 'System Image Generation Utility.' This may be used to determine the system requirements for a user’s application. Sysgen then selects the appropriate tools from the 'development platform' translates and binds them into an 'Application Binary Image.' This image can then be transferred to ROM or RAM.

Dynamic Binding

Alternatively, it is possible to load tools by using 'dynamic binding.' This feature allows tools to be loaded into memory only as and when they are required. When individual tools are 'called' the kernel tool loader loads them into memory, linking them into other resources on the system

While waiting to be loaded into memory, the stored templates typically exist in Virtual Processor Code (VP for short). Translation of VP into native code occurs only as tools are loaded from store into memory. Numerous different types of 'call' may be used to load the required tools from store, each designed to enhance the efficiency of the translation process. These are different types of *qcall*, or 'quick call' of which the principal variants are as follows:

- **Non-virtual**

This means that the tool was loaded and bound when the object referencing it was loaded. It will remain available in local memory at least as long as the referencing object is in memory. It will not be relocated in memory whilst the caller is present.

- **Virtual**

This means that the required tool need not be in local memory at the time it is required. If the tool is not available in local memory the tool is loaded and bound before it is available for use. On exit from the tool its memory space may be deallocated by the kernel in order to free local memory space. While the tool is referenced it will remain in memory, however it cannot be assumed that this will be the case when the tool is no longer referenced.

- **Semi-virtual (*VIRTUAL+FIXUP*)**

This means that, in the same way as a virtual tool, it is only loaded on the first call to it, and not when the caller is loaded, but is then treated as non-virtual. i.e. it remains in memory until no longer referenced by the calling tool.

- **Embeddable**

This means that the entire body of the subtool is inserted in-line. Please note that embedding should be used sparingly. It is normally used to qcall a tool that is typically a quite small number of instructions.

For more information, please see section 7.3.1.

Before being stored on disk, tools are compiled or assembled into VP binary code. This uses a bit efficient instruction set to maximise code density in store. Consequently, an application in VP code generally requires less storage space than the native code into which it will be translated, often by a ratio of between 1.4:1 and 2:1. Since there are fewer bytes to load for a VP code application than for a native code version, the load time is significantly reduced.

Software written or compiled to VP specification may be examined by the Data Flow Analysis Utility (DFA). The DFA is a tool run from the shell to ensure that code complies with VP specification and hence is genuinely portable between intent implementations. The DFA performs a static analysis of the code, examining it without running it. (The DFA is also able to use its analysis of data flow anomalies to perform optimisations, and to detect coding and design errors in the software.)

3.1.2 Endian-Independence

Endianness refers to the order in which all byte-representations are stored in memory, e.g. in the integers and long integers. A little endian processor will place the little end, or least significant byte, in the lowest address. A big endian will place the big end, or most significant byte, in the lowest address. For convenience, the VP code memory model is defined to be "little endian," the low address accessing the least significant byte of the integer, long integer, or float, etc.

On "big endian" processors, an 'address transformation' technique is usually performed by the translator to give the effect of little endian addressing. This technique is only required when loading any quantity smaller than 32 bits, as only the words are stored in natural byte order. The order that VP accesses the bytes or shorts in an integer is reversed by inverting the low 2 address bits before the access occurs. As a result of 'address transformation,' applications may run slightly more slowly on big-endian machines. This issue may also have an impact upon the writing of device drivers, where it may be necessary to take special measures, for example, in such areas as byte swapping data from a filesystem or network.

3.1.3 Translation versus Interpretation

Translation should not be confused with interpretation, another widely used means executing a portable binary representation. Interpretation is favoured by numerous systems, including many script languages, and virtual binary systems such as BCPL cintcode, Smalltalk byte codes and Java™ byte codes.

A platform using translation initially takes a little longer to load code into memory but is afterwards able to operate at full native speed. An interpreted system, however, suffers an ongoing run time penalty. In effect, an interpreted system uses a form of cross-language glossary prior to executing any piece of code it encounters, however many times it may already have encountered and interpreted that same piece of code. As a consequence, interpreted systems tend to be inefficient, often running 10 to 100 times more slowly than would a direct execution.

3.1.4 Translation versus Direct Execution upon Hardware

It has been occasionally suggested that hardware could be devised that would permit VP to execute directly upon it, since this could occur at full speed without incurring the load time penalty for translation. However, this would seriously compromise flexibility. Optimising Code generators or careful hand coding would be needed to ensure a match between the code generated and the hardware provided. Even trivial updates and alterations to the hardware would probably force the range of instructions to be reordered or otherwise adjusted.

3.1.5 Available Translators

Tao provides a range of translators designed to convert a tool in VP code into a native tool. Some translators, however, are used to perform validity checks upon an input tool. In such cases there will be no output tool. In theory, it would also be possible to implement translators that would convert one VP tool into another, effecting some kind of transformation.

3.2 CPU Isolation Interface

When porting to a new processor, the CPU Isolation Interface (CII) must be written. The CII is a collective term for the tools that provide an interface between the kernel and the processor, and into which encapsulates the behaviour of the processor upon which Elate is to be run. The kernel itself requires no rewriting, the CII and PII (Platform Isolation Interface) acting as containers for all non-portable functions.

Among the functions included in the CII includes tools which interface with the instruction set of the individual processor, tools which mediate between the scheduler and the processor during task switches, and stack manipulation tools.

3.3 Processor Specific Libraries

Like its counterparts in other operating systems, the int**e**nt library is comprised of a collection of routines, related in terms of their purpose. However, in the case of Elate, these routines are not placed together in a single file. Instead, each library function is contained in a separate tool, which are all linked individually. The library is a group of tools in a directory tree.

There occasionally occurs a need for an operation that VP does not support, or that can be executed more efficiently by native code. A games console, for example, might include a card for producing polygon-based graphics. Under these circumstances, int**e**nt might operate more efficiently by exploiting this piece of specialist hardware than by running its own facilities for graphic polygons. In this case, the int**e**nt system might qcall the native functions connected with the graphics card. In another case, complicated mathematical functions, or functions designed to perform black memory transfers, might be available through native code.

4. Platform Ports

4.1 Platform Isolation Layer

The Platform Isolation Layer (PIL), rather than being a structure in its own right, is a convenient collective term for all of the code which is specific to the platform, and thus distinguishable from the portable main bulk of the operating system. The PIL includes the Platform Isolation Interface and the platform specific device drivers.

4.1.1 Platform Isolation Interface

The PII is a collection of functions that allow the kernel and the device drivers to gain access to certain features offered by underlying hardware or software. The PII will differ for each Elate system. When no hosted environment is present, the PII would be a simple mechanism for directly accessing hardware. On a hosted platform, the PII will interface with the host Operating System (OS). If a particular platform has no use for certain PII functions, these may not be implemented, or may be partially implemented. For example, functions for allocating or freeing memory may simply return errors if all system memory has already been allocated, as in the case of a simple plug-in board, or an embedded system.

To the kernel, the PII provides functions that include system startup and shutdown, for allocating and freeing memory, for locking and unlocking memory and for setting up exception handlers. The device drivers can access functions for setting and unsetting interrupt handlers, for mapping and unmapping physical address ranges into process address spaces, for getting physical addresses of logical addresses, for calling a host OS. Both the kernel and the device drivers are offered functions for enabling and disabling hardware interrupts, and tools for power management.

See *The Platform Isolation Interface Reference Manual* included with the release documentation for further details.

4.2 Using the Java Native Interface on a Hosted Platform

The Java Native Interface (JNI) is a standard programming interface for writing Java native methods and using a Java virtual machine from native applications. JNI is intended to provide binary compatibility of native method libraries across all Java virtual machine implementations on a given platform.

Under the normal intent model a Java application running under the intent portability layer has the ability to have several threads accessing a non-Java intent library through the Java Native Interface (JNI). However, in the case of accessing a library located upon the host operating system that intent would currently be running upon, JNI can be used to provide access to native host libraries.

The Java Native Interface has been implemented to specification to allow either native programs to access Java methods (or vice versa where appropriate). For example a C program may access a Java function, which may then use JNI in turn to access a native library function, thereby permitting the use of otherwise unavailable functionality. Provided that the native code has been written in C or C++ there should be no obstacle to using JNI. As far as a Java application programmer is concerned the model is completely transparent.

In order to support JNI in a given host, it is necessary to compile the jni.c files for that particular platform. It is also necessary to create two .h files, which provide the JNI mechanisms with information about the hosted platform:

- *host-md.h*
- *host-hlp.h*

The first of these essentially defines the primitive types such as 8, 16, 32, and 64 bit integers. The second defines a set of OS independent functions for allocating memory, starting processes, using mutexes and so on.

4.3 Device Drivers

Device Drivers provide a range of services from interfacing to hardware, such as I/O ports, to providing software only services, such as runtime interfaces to host operating systems. A Device Driver is a software component, which offers a generic Application Programmer Interface (API) to a device family. The API remains the same for each device of that family but each Device Driver includes platform and processor specific code. In this way, the implementation of any Device Driver remains transparent to the application, which has no knowledge of how the device provides its services. The Elate operating system supports an object based programming style, so that a Device Driver within Elate is programmed as an object.

For an application to use a Device Driver, it must be mounted. Elate provides various ways of easily doing this. All the procedures place a reference to the Device Driver in a look-up table called a Mount List and a <*name*> that is associated with this reference. In this way, a new Device Driver can be written, and can take the place of a previous version in the Mount List, and as long as it is given the same <*name*> as the previous driver, the application is able to access it without being aware of the change.

4.3.1 'Direct call' versus Pollin g

Device driver code is normally executed as a subroutine of the calling process, rather than operating by sending and receiving mail messages. The device drivers use a 'direct call system.' This runs the device drivers as subroutines of the application. As a consequence, it is less often necessary for the kernel to switch tasks or perform memory management.

A number of tools and objects exist to simplify the implementation of new device drivers. See *The Device Driver Design Guide* for further details on this and other issues.

5. Platform Isolation Interface (PII)

5.1 PII Services

The PII provides services that can be grouped into three categories:

1) Services for the kernel
2) Services for device drivers
3) Services for device drivers and the kernel

The majority of PII tools are written in VP assembly language. PIIs can be divided into two broad types, hosted and non-hosted. ‘Hosted’ is where Elate permits intent runs on top of another operating system. In this case the PII tools very often call the underlying OS to carry out a particular function e.g. lock memory.

A diagram illustrating the relationship between the PII and other subsystems is shown below:

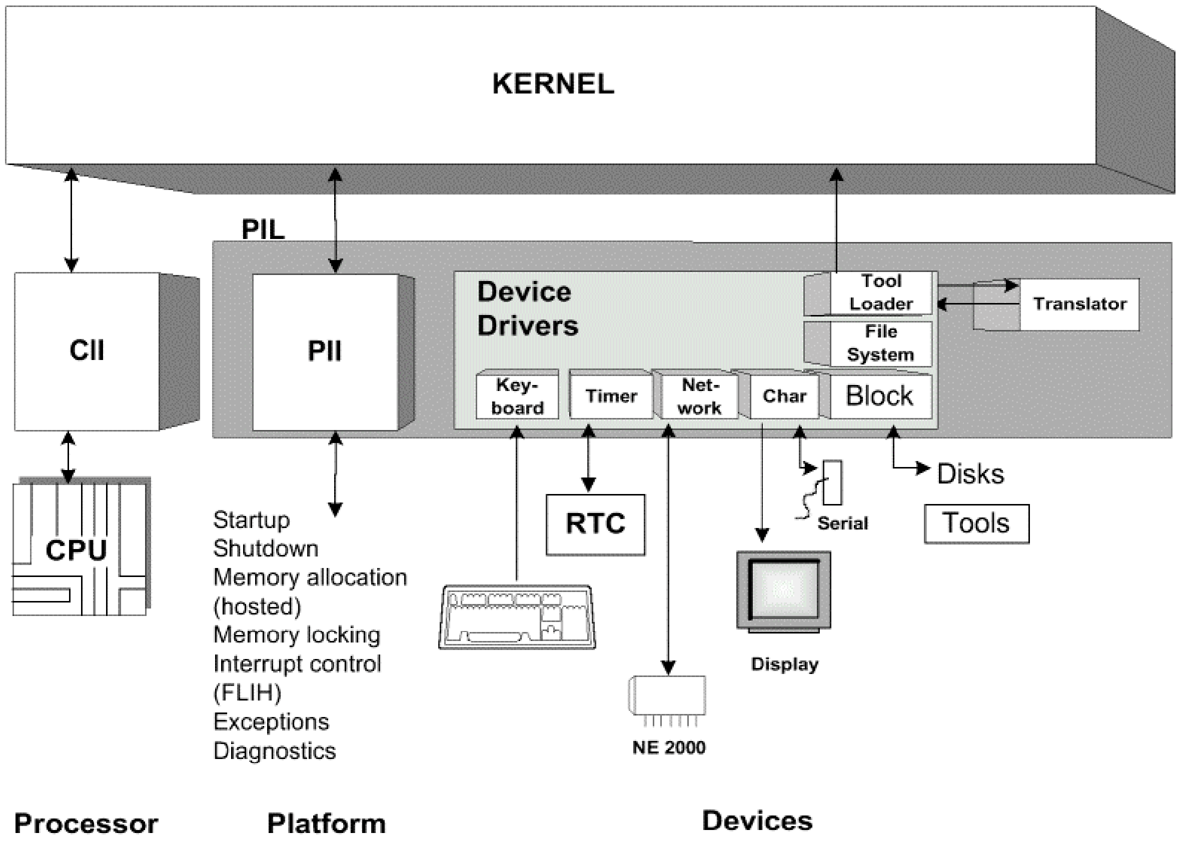

Figure 2. Platform Isolation Layer, device drivers and kernel.

5.1.1 Kernel services provided by the PII

- System start-up tools
- Tools for locking/unlocking memory
- Tools for memory allocation
- Tools for setting exception handlers

5.1.2 Device driver services provided by the PII

- Mapping and unmapping of physical address ranges into process address spaces
- Setting and unsetting of interrupt handlers
- Getting physical addresses of logical addresses

5.1.3 Kernel and device driver services provided by the PII

- ♦ Returning the address of the host parameter string
- ♦ Enabling and disabling interrupts

5.2 PII Tools

Below is a list of the most common PII tools, with a brief description of their function. For more information refer to the *Platform Isolation Interface Reference Manual*, within the system release at *sys/pii/.* If you are about to write a PII please also refer to the appropriate PII Construction Guide (hosted or non-hosted), which also reside within the *sys/pii/* directory.

5.2.1 System Startup and Shutdown

- *sys/pii/boot* - the boot tool is responsible for booting the system into a 'runnable' state
- *sys/pii/shutdown* - shutdown and release all allocated resources

5.2.2 Information Provision

- *sys/pii/info* - returns a PII type and a pointer to a PII specific block of memory
- *sys/pii/stksize* - returns minimum amount of extra stack space required for the PII to run

5.2.3 Memory Allocation (hosted only)

- *sys/pii/alloc* - calls host OS to allocate memory
- *sys/pii/free* - calls host OS to free memory

5.2.4 Memory Management

- *sys/pii/lock* - locks a memory region so it's not paged out to disk
- *sys/pii/unlock* - unlocks a memory region, which becomes eligible for paging out to disk
- *sys/pii/map* - maps a physical address space to the address space of an Elate process
- *sys/pii/unmap* - unmaps a process address space so that memory resources are freed
- *sys/pii/get_phys* - gets the physical address of a memory region
- *sys/pii/makeexec* - makes a memory region executable
- *sys/kn/mem/gwc* - Default memory manager

5.2.5 Interrupt Management

- *sys/pii/setint* - allocates an ISR (SLIH) to an IRQ (supports multiple ISRs on one IRQ)
- *sys/pii/unsetint* - removes ISR from an IRQ
- *sys/pii/int_dis* - disables the specified IRQ
- *sys/pii/int_en* - enables the specified IRQ
- *sys/pii/int_off* - disables all hardware interrupts
- *sys/pii/int_on* - enables all hardware interrupts, except those specifically disabled
- *sys/pii/int_restore* - restores the interrupt state to that returned by *sys/pii/int_off*
- *sys/pii/sched_op* - cause scheduler operation to be carried out ASAP (FLIH returns to scheduler)
- *sys/pii/halt* - halts Elate allowing host OS to take control

5.2.6 Exception Management

- *sys/pii/setexc* - allocate exception handler routine to handle specified exception
- *sys/pii/unsetexc* - deallocate exception handler from specified exception (uses handle returned by *sys/pii/setexc*)

5.2.7 Host OS Specific

- *sys/pii/opentool* - opens and binds host specific code to an Elate tool
- *sys/pii/closetool* - deallocate resources associated with host tool binding
- *sys/pii/hostparms* - return address of host parameter string
- *sys/pii/stkvars* - setup platform specific stack on hosted PIIs
- *sys/pii/threadevent* - notification of thread creation/destruction in hosted environment

5.2.8 Process Synchronisation

- *sys/pii/swap* - swap a non-zero integer and return result

5.2.9 Diagnostic Functions

- *sys/pii/odata* - output diagnostic string to host system (platform specific)
- *sys/pii/ktrace* - output diagnostic string with numerical data (calls odata, portable)
- *sys/pii/trdata* - hex dump of memory to trace system (calls odata, portable)

6. Device Drivers

6.1 Introduction to Device Drivers

Device drivers are comprised of classes written in VP that control various hardware devices directly. To carry out certain functions the drivers will call upon the services provided by the PII, for example, loading interrupt service routines is handled by the PII.

The device driver hierarchy is strictly class based, with all device drivers inheriting from *dev/class.* This means that drivers for a device family can quickly be created by subclassing, the subclasses having hardware specific details where required. The diagram below illustrates this:

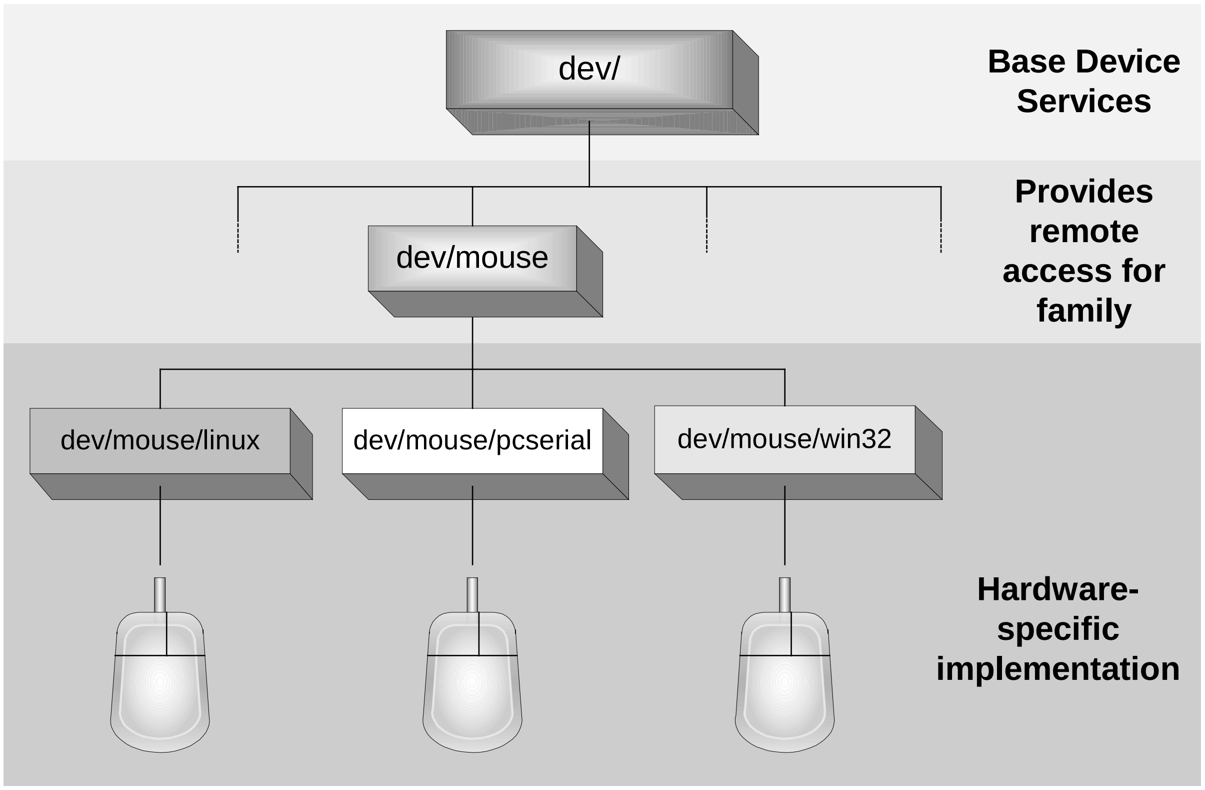

Figure 3. Device driver hierarchy

This class based approach has additional benefits; Elate device drivers have a very well defined set of methods that need to be implemented, this in turn means that the device driver API is consistent and easy to use. It is also relatively easy to extend existing drivers or develop new drivers. Because there is no distinction between the kernel or user mode in Elate, device drivers can be called directly from user applications. It is therefore possible to start, stop, load and unload drivers at run-time. Note that unlike some operating systems the Elate device drivers are not compiled into the kernel.

Elate device drivers are usually interrupt driven. As will be explained in more detail later, an interrupt service routine (ISR) is installed as part of the driver *init* method.

Alternatively, I/O can be achieved by utilising Elate's extensive libraries, such as the ANSI C library. These libraries will in turn call device drivers for low level I/O as required.

6.2 Writing Device Drivers

Where programmers need to write new device drivers to support special or new hardware, they are advised to read '*The Device Driver Design Guide*.' This document has extensive details on how to write drivers. Here we will only consider the main issues that arise in developing drivers.

These issues are discussed below:

6.2.1 Memory Mapping

As mentioned above, the PII provides native tools that are responsible for mapping areas of physical memory into an Elate process's address space. Device driver programmers will require the ability to map between physical and logical memory.

6.2.2 Access to Input/Output (I/O) Ports

Access to hardware I/O ports should generally be achieved through use of the *sys/cii/ioin* and *sys/cii/ioout* macros. The *sys/cii/natst* and *sys/cii/natld* macros may also be used to access memory mapped IO ports and memory, in cases where direct access is required without memory address transformation.

6.2.3 Exclusive Software Resource Access

Typically various data and information about a specific device is held in the device object's instance data (please refer to the '*Object Based Programming Guide'* for more information on objects and instance data). Each device driver contains a mutex to ensure data integrity is preserved in situations where multiple processes could access the driver.

6.2.4 Page Out Prevention

It is important that certain routines, e.g. interrupt handlers are not paged out to disk. This could potentially occur when Elate runs on a host OS that utilises paging (virtual memory management). In order to prevent this the tools *sys/kn/mem/lock* and *sys/kn/mem/unlock* can be used. Note that these routines will call the necessary native tools in the PII i.e. *sys/pii/lock* and *sys/pii/unlock*.

6.2.5 Exclusive Hardware Resource Access

On initialisation it may be advisable to lock the I/O registers of a device. This can be achieved by using the tool *dev/lockio*. This tool can also be used to 'book' an Interrupt Request ID.

6.2.6 Device Driver Read/Write Policy

It is also important to decide on the device driver policy that will be implemented. Currently there are two available policies:

- blocking (aka synchronous)
- non-blocking (aka asynchronous)

These policy types are discussed in more detail below.

6.3 Device Driver Methods

As mentioned above the method API for a device driver is very well defined. Below are listed some typical methods. Drivers may not support all these methods; for example, a keyboard driver would not have an implementation for the *write* method, for obvious reasons.

Methods implemented by all device types:

- *_init*
- *_alias*
- *open*
- *_deinit*
- *info*

These methods are documented more completely in *dev/api.html.*
Additional methods implemented for character devices:

- *close*
- *reada*
- *statusa*
- *ioctlex*
- *status*
- *flush*
- *getflags*
- *reference*
- *writea*
- *cancel*
- *read write*
- *sync*
- *status*
- *setflags*

Other methods are easily added as required for specific devices, e.g. a CD-ROM driver might have the following additional methods:

- *cdeject*
- *cdplaytrack*
- *cdload*

6.4 Device Driver Allocator/ De-allocator Tools

Device drivers have two special tools called *_new* and *_delete*. These tools are responsible for allocating and de-allocating memory for the objects respectively and providing an object pointer which can then be used to invoke the device driver object method code.

Example code is given below:

```
tool 'dev/example/_new'
      ent - : p0                          ; return instance pointer
      cpy.i MH_SIZE+DEV_SZ,i0
      qcall sys/kn/mem/allocdef,(i0:p0,i~)
      if.p p0 != NULL
            add.p MH_SIZE,p0
            refclass p0,dev/example/class
      endif
      ret
toolend


tool 'dev/example/_delete'
      ent p0 : -
      derefclass p0
      sub.p MH_SIZE,p0
      qcall sys/kn/mem/free,(p0 : -)
      ret
toolend
```

6.5 The Anatomy of a Typical _init Method

Normally a driver *_init* method would perform the following tasks:

- ♦ initialise instance variable and override these with values from command line parameters if present
- ♦ initialise a mutex in preparation for locking instance data in various methods (using *sys/kn/mtx/init*)
- ♦ lock I/O address range for device using *dev/lockio*
- ♦ book IRQ line for use by the device, also using *dev/lockio*

- load the device ISR with *dev/_loadisr*

- set-up the hardware device by:
  1) disabling interrupts with *sys/pii/int_off*
  2) writing to the hardware or check status using *dev/out* and d*ev/in* tools
  3) switching interrupt back on using *sys/pii/int_restore*

Note that the *_init* method will often need to perform superclass initialisation before the subclass can be instantiated. If for some reason the superclass is initialised successfully, but for some reason the subclass object fails to initialise, then this should be trapped and the superclass object deinitialised.

6.6 Anatomy of a Typical _deinit Method

The *_deinit* method typically reverses the procedure carried out by the *_init* method. This involves deallocating any buffers that may have been allocated, deinitialising mutexes, disabling interrupts, unlocking memory areas and unloading ISRs. In addition the subclass object will need to ensure that the parentclass object is deinitialised.

6.7 The Anatomy of a Typical reada Method

A simplified *reada* method would:

1) lock instance data with *sys/kn/mtx/lock*
2) If data is available, return buffered data, else queue request
3) unlock instance data with *sys/kn/mtx/unlock*

6.8 The ISR

If the device driver is interrupt driven, then when the device driver is initialised an ISR is loaded. The specific operations carried out by this routine are device dependent. This routine is also known as a Second Level Interrupt Handler (SLIH). It is called by the First Level Interrupt Handler (FLIH). The FLIH is normally coded in native assembler and is part of the PII, e.g. for the PC Standalone platform the FLIH is found in the tool *sys/pii/pcstand/flih*. The tools *sys/pii/setint* and *sys/pii/unsetint* are actually responsible for attaching and detaching SLIHs to the FLIH, although *dev/loadisr* provides a convenient alternative for device driver work. Note that multiple SLIHs on one IRQ are supported for the situation where several devices share an interrupt ID. The PII chains these handlers, so that once an IRQ has occurred each SLIH for that interrupt ID is called in turn. The SLIH is coded so that it can determine whether it is actually meant to handle an interrupt from that device or not. If it does handle the interrupt it may need to wake up any process waiting on this event.

ISR operation is roughly as follows:

1) Interrupt. If there is a queued AIO request then:
   2) Copy data to user buffer and make a callback to the user's process to complete the AIO operation.

Else:

3) Buffer the data.

If the second and third steps are going to take significant time (for example, a device driver needs to do byte swapping on a large buffer) then the ISR would make a callback into its own high priority helper process which would carry out the work with interrupts re-enabled.

6.9 Blocking and Asynchronous read/write Policies

There are generally two types of I/O policy in Elate:

- blocking
- asynchronous (AIO)

6.9.1 Blocking

Here the device driver blocks until the I/O operation completes. Note that with both I/O policies the driver code is 'ncalled' from the application and thus generally runs in the context of the application's process. The blocked process can then be woken by the ISR to complete the I/O using a call to *sys/kn/int/proc/wake*.

6.9.2 Asynchronous

There are several parts to this system:

- calling application
- notification mechanism
    - callbacks
    - event flags
    - signals
- AIO structure
- device driver
- interrupt service routine

6.9.3 Notification mechanism

The purpose of this is to provide a mechanism whereby the calling application can be made aware that an I/O operation has now completed and that the application code to process this can now be run. This section discusses callbacks as this is a popular notification mechanism.

A callback handler (subroutine) can be written at the application level to process the result of an I/O operation. The callback routine does whatever the application needs to do once the I/O operation has been completed by the underlying layers. For example, in the AVE, if data becomes available from the keyboard, the callback will dispatch a keyboard event to the target AVO.

6.9.4 AIO structure

This structure is used to store important information about the AIO operation. Each AIO operation requires an AIO block, also the AIO block must only be reused after the AIO has completed.

AIO blocks make it possible for the driver to manage a number of simultaneous outstanding I/O operations.

The structure of the AIO block is described in the device driver documentation at *dev/api.html*

6.9.5 Interrupt Service Routine

The ISR performs initial processing on the incoming interrupt. Before returning, the ISR will call *dev/iocomplete* to complete the I/O operation, using a pointer to the AIO block of the calling application's process as a parameter. This flags the callback routine registered by *dev/ioprepcallback* to be processed by the kernel.

6.9.6 Sequence of Operations for Asynchronous I/O

The application calls a reada or writea method on the device object. The driver code returns to the application immediately having initiated an I/O operation. The application code would typically sleep at this point, effectively waiting for completion of the I/O without using processor resource.

When the I/O operation is completed by the hardware an interrupt occurs, which is processed by the installed ISR. The ISR will call *dev/iocomplete* when the initial interrupt processing is done. *dev/iocomplete* flags a callback to be processed by the kernel (in this case it does this by calling sys/kn/callback/set). The callback is registered to run at the application process level, although it is possible to register the callback to run at the device driver process level (if there is one). This is achieved using *dev/ioprepdevcallback.*

6.9.7 Device Driver Method Implementation

It is important to note that the device driver developer only needs to code asynchronous method calls when implementing a new driver. This is because the driver base class dev/class fully implements methods such as read, write, status etc. by using calls to reada, writea, statusa, which are implemented further down the class hierarchy. Further, the base class adds wrapper code around these calls to the asynchronous methods to implement the blocking nature of the read, write and status methods (where required).

As stated earlier, this leaves the device driver developer free to concentrate on developing the asynchronous methods, without having to worry about handling the 'non-asynchronous' methods.

6.9.8 Example Code

```
Application

dev/ioprepcallback,(aio_ptr, callback_handler, callbackdata_ptr)
ncall
dev_ptr,reada,(dev_ptr,dev_handle,callbackdata_ptr,aio_ptr,bytes_to_read:fl
ags)

loop
      qcall sys/kn/proc/sleep,(-1.l:i~)
endloop

Application callback handler

callback_handler:
      ; p0 = aio_ptr
      ; p1 = callbackdata_handler

      ent p0 p1:-
      …

      qcall dev/iogetresp,(aio_ptr:status)      ; get final status of
operation
      …     ; Start off next asynchronous read operation
      qcall dev/ioprepcallback, (aio_ptr, callback_handler,
callbackdata_ptr)
      ncall
dev_ptr,reada,(dev_ptr,dev_handle,callbackdata_ptr,aio_ptr,bytes_to_read:fl
ags)
      …
      ret

Interrupt service routine

handle incoming event
…
dev/iocomplete,(aio_ptr, status:return_status)
ret
```

6.9.9 Extensive Interrupt Processing

If the ISR needs to do more extensive processing of the incoming interrupt then an alternative approach is necessary. This is because in Elate interrupts are disabled while in ISRs. This could potentially result in interrupts not being processed in a timely manner. The Elate solution for handling nested interrupts is to run the code to handle the interrupt in a separate, high priority process. This process needs to be of a high priority to ensure timely execution by the kernel i.e. ensures that interrupts are not missed.

The code to create the high priority process is actually inherited from the device driver base class *dev/class.asm*. The process created on demand by ncalling the *_devproc* method. This provides a high priority context in which a callback handler registered at the device driver level can be executed. This driver level callback handler could perform a *dev/iocomplete*, flagging up a callback handler registered at the application level to execute, within the application's context.

6.10 Starting and Stopping Device Drivers

There are three ways to start device drivers:

1) using *.obj* in a sysgen .sys file
2) using the shell command *devstart*
3) under program control

All three techniques essentially do the same thing:

- create a device driver object
- initialise the device driver object
- add the device driver object to the mount table

Note that at any stage the device driver mount table can be displayed by using the shell command (assuming that the system has a shell):

```
$ devinfo
```

If information on a specific driver is required, *devinfo* can be used in conjunction with the driver name:

```
$ devinfo /device/mouse
```

Note that the $ character is merely the shell prompt and is not part of the command.

6.10.1 Using Sysgen to Install Device Drivers

To illustrate this a snippet of code from a sysgen file is presented:

```
                  :
.obj(newtool="dev/timer/pc/_new",mountstr="/device/timer",cmdline="timer  -
r6");
.obj(newtool="dev/blk/pcide/_new",mountstr="/device/ide1",cmdline="ide1   -
p$1f0 -i14");
.obj(newtool="dev/blk/partit/_new",mountstr="/device/partition",cmdline="pa
rt -b/device/ide1/0");
.obj(newtool="dev/fs/fat/_new",mountstr="/device/fatfs",cmdline="fatfs    -
b/device/partition/0");
      :
.obj(newtool="dev/fs/fat/_new",mountstr="/device/fatfdd",cmdline="fatfdd  -
b/device/fdd/0");
      :
.obj(newtool="dev/trace/elate/_new",mountstr="/device/trace",cmdline="trace
-d/trace.log");
.obj(newtool="dev/trace/elate/_new",mountstr="/device/error",cmdline="error
-d/error.log");
                  :
```

Looking at the first line of this sysgen script sequence we can see that the timer allocator tool is being referenced i.e. *dev/timer/pc/_new*. In the next part of the statement a string is allocated to the device. The mount string is the name that becomes associated with the device driver for the purposes of the mount table. The mount table can be displayed by using the shell command *$ devinfo*. Note that two instances of the same driver class can be installed twice if two different mount strings are specified, e.g.:

```
.obj(newtool="dev/trace/elate/_new",mountstr="/device/trace",cmdline="trace
-d/trace.log");
.obj(newtool="dev/trace/elate/_new",mountstr="/device/error",cmdline="error
-d/error.log");
```

In this case the trace device driver is used to provide a trace device and an error device.

Note also that any command line parameters required are also specified using the cmdline facility.

6.10.2 Using devstart

To start a device driver from the command line you will need to use *devstart* followed by the required parameters. The basic syntax is *devstart* <device name> <device driver> [other parameters] e.g.:

```
$ devstart /device/mouse dev/mouse/pcserial
```

To unload this device driver you would type:

```
$ devstop /device/mouse
```

Note that you need to specify the driver name, as there may be several device instances of the same type sharing the same driver object.

If additional parameters are not provided then the driver will use its built-in defaults.

6.10.3 Loading Device Drivers From Software

This can be done by using the following sequence of instructions:

1) call the device driver allocator *tool _new*
2) invoke the driver object *_init* method
3) call *sys/kn/dev/mount* to add the device to the mount table

6.10.4 Accessing Drivers From an Application

Assuming the device driver has already been successfully loaded via use of the *devstart* command (say from within a script) or via an *.obj* command (from a sys file), the procedure for accessing the driver from your application is as follows:

1) Use *sys/kn/dev/lookup*
2) Open the device
3) Invoke methods on device as required by the application
4) Close the device

Each of these steps is explained in more detail below:

1) ***sys/kn/dev/lookup*** - takes as its input a string. This string is the logical name of the device. Examples of logical names include:

```
      /device/loader
      /device/error
      /device/null
      /device/rwfs
      ….
```

These logical names appear in the mount table, which as noted above in this document can be viewed by the *devinfo* shell command. *sys/kn/dev/lookup* will scan through the mount table looking for the best match to the string you provide. *sys/kn/dev/lookup* will then return with two pointers:

- A pointer to the device driver instance (or errno if it fails)
- A pointer to a residual string

The instance pointer is subsequently used to '*ncall*' device methods as required in the application. The residual string is the difference between the string provided by the application and the closest matching logical name found in the mount table. For example, assume the device driver defined by *dev/example/class.asm* is loaded and mounted as */device/example*. If the programmer then performs a *sys/kn/dev/lookup* providing *dev/example/foobar* as the input string then the residual string would be 'foobar'. For an exact match the residual string will be blank; the caller can check for this if an exact match is required.

The purpose of the residual string is to provide access to a resource within the driver. What this means in practice depends on the actual device and is discussed in the user documentation for the associated driver. As an example, the Sejin keyboard driver uses the residual string as a way of selecting the mode of operation of the keyboard. In the filesystem the residual string is used as the filename for the file.

On a practical note, *sys/kn/dev/lookup* is unlikely to fail, as it is very difficult for it not to find a partial match in most cases. The residual string should therefore always be checked by the programmer. It is also advisable to check the device family by invoking the info method.

2) **open** – this method is invoked to open the device and return a handle to the relevant resource. This handle is typically used in subsequent *ncalls* to the driver methods. The returned handle should be checked by the programmer to ensure the device opened successfully. The *boolerrno* and *iferrno* macros are particularly useful for this purpose. The device may fail to open because another application is already using it. Another point worth noting is that some device drivers can actually have one instance with multiple handles. A good example is the floppy disk controller, which can have one instance with four handles, one handle for each of the drives it can control.

3) **Invoke methods as required** – for example:

```
      ncall p0,info,(p0,p1,p2,i0:i0)
```

4) **close** – this method closes the device (but does not delete the device from memory or from the mount table). Example call:

```
      ncall p0,close,(p0,p1:i0)
```

Applications are responsible for closing devices when they exit, even in exceptional situations. The library functions lib/open, close, read etc provide an interface to ensure that open device devide handles are closed on exit.

6.10.5 Device Families

The Elate device driver architecture currently supports the following device driver families:

```
      structure
      byte  DF_RESERVED ;Not used
      byte  DF_UNDEFINED;Unknown or undefined devices
      byte  DF_BLOCK    ;Block devices (e.g. HDD)
      byte  DF_CHAR     ;Character devices (e.g. Serial)
      byte  DF_SOUND    ;Sound devices (e.g. WAV)
      byte  DF_KEYBOARD ;Raw keyboard devices (e.g. Keypad)
      byte  DF_POINTER  ;Pointer devices (e.g. Mouse)
      byte  DF_FILESYS  ;Filesystem devices (e.g. FAT)
      byte  DF_TABLET   ;Tablet devices (e.g. penpad)
      byte  DF_PROTOCOL ;Protocol devices (e.g. TCP/IP)
      byte  DF_NETWORK  ;Network devices (e.g. Ethernet Adapter)
      byte  DF_MESSENGER;Messenger driver
      byte  DF_LINK     ;Link devices
      byte  DF_GRAPHIC  ;Graphic devices (now superseded by DF_PLANE)
      byte  DF_AVE      ;Audio Visual Environment devices
      byte  DF_3DGRF    ;3-D grf services (now superseded by DF_PLANE)
      byte  DF_TIMER    ;RTC timer devices
      byte  DF_JOYSTICK ;joystick device
      byte  DF_CPLOADER ;coprocessor loader
      byte  DF_CPDISPATCHER   ;coprocessor dispatcher
      byte  DF_PLANE    ;plane driver
      byte  DF_MSGPIPE  ;Message-based pipe driver
```

For more details, examine the file *lang/asm/include/devices.inc.* The *info* method returns the device family as part of its information packet.

The command *devinfo* uses the *info* method of the driver to return this information:

```
19990823:/$ devinfo /device/rwfs
/device/rwfs:
  Family: filesystem
  Caching Longname Driver Version 1.24 over Win32 Filesystem Driver Version
1.22
19990823:/$
```

6.11 Available Device Drivers

6.11.1 Device Driver Types

Device drivers can be grouped into two main categories:

- ♦ hardware dependent – control hardware directly i.e. device or platform specific
- ♦ generic – portable across different platforms

The relationship between generic and hardware dependent drivers is illustrated in the following diagram:

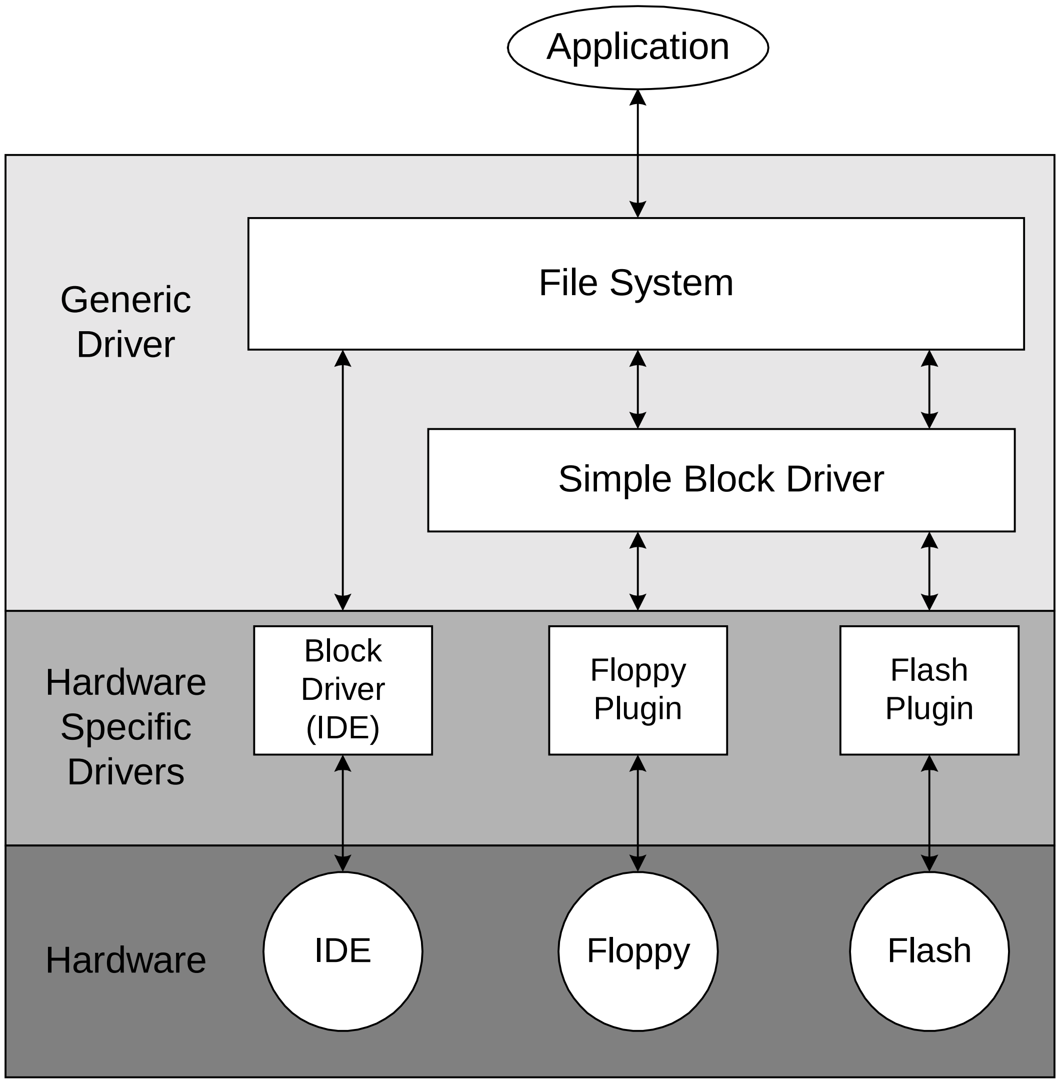

Figure 4. Generic and hardware dependent drivers.

Drivers can be further classified according to the type of device they control, the main categories being:

- character
- sound
- network
- keyboard
- link
- protocol
- timer
- block
- pointer
- graphic
- filesystem
- messenger

6.11.2 Hardware Specific Drivers

For example, consider the device drivers available for a stand-alone PC system (i.e. no host OS). Some of the currently available drivers include:

Hardware device	Driver type	Description
Real-time clock	Timer	Uses RTC periodic interrupt
PS2 keyboard	Keyboard	
Character display	Character	80x25 text mode display driver
IDE controller	Block	
SCSI controller AIC 78xx	Block	Adaptec AHA 2940 SCSI
Floppy disk controller	Block	

PS2 mouse	Pointer	
Serial mouse	Pointer	
Serial port	Character	16550 UART
NE2000 controller	Network	Ethernet adapter device driver

Note there are many other drivers available for other platforms such as:

- ♦ Windows hosted device drivers
- ♦ Linux hosted device drivers
- ♦ Hosted device drivers for other hosted platforms
- ♦ Device drivers for various single board computers (SBCs), set-top boxes and network computers

Typically, a hosted version of a device driver will simply make a call to the underlying OS in order to carry out a device related activity.

6.11.3 Generic Device Drivers

Elate also includes a number of generic device drivers, which are portable across many different platforms. They normally use the service of an underlying hardware specific driver. Elate's generic drivers are discussed in more detail in the remainder of this section.

- NULL device (character)

- Trace device (character)

This device provides system tracing facilities.

- Keyboard cooker (character)

Takes raw key up and key down codes from the keyboard device driver and cooks these to an appropriate character code.

- Tool loader

The tool loader uses an underlying filesystem driver. Its job is to find tools on disk and load them, calling the translator if required.

- File name mapper (filesystem)

Uses an underlying filesystem driver. This is used where the system is running hosted by another OS. The mapper provides a mapping between Elate format filenames and those of the underlying filesystem. For example when Elate is running DOS hosted, the mapper would be required to map Elate long filenames to the 8.3 format of the underlying DOS filesystem. The mapper handles issues such as:

- ♦ file name length
- ♦ case sensitivity
- ♦ case preservation

An application of the file name mapper driver is shown in the following diagram:

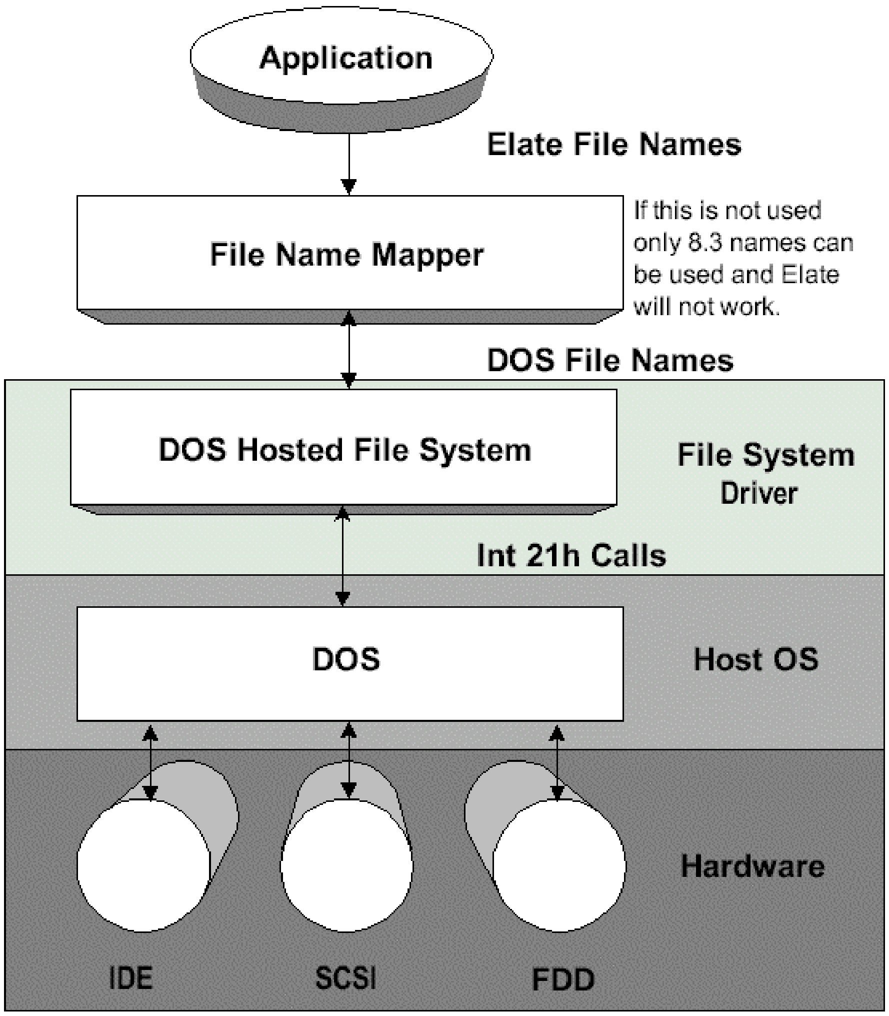

Figure 5. File name mapper.

♦ ZIP filesystem (filesystem)

Enables a compressed or uncompressed Zip file to appear to Elate as a read-only filesystem. The Zip file may be in ROM or as a file on an underlying filesystem.

♦ File block driver (block)

Uses an underlying filesystem driver. Enables a file to appear as a block driver. Can also provide services to the Elate filesystem.

♦ Partition driver (block)

Uses an underlying block driver. Manages partitions on a disk by making each partition appear as a separate block device.

♦ Pipe (character)

Named and un-named pipes. Pipes are local to Elate.

• Messenger (messenger)

Uses an underlying link driver. Used to communicate between Elate processes that are running on different physical CPUs.

- Character link (link)

Uses an underlying character driver. Used in Elate messaging described above.

- PPP (network)

Uses an underlying modem driver. Implements the Point to Point Protocol used to obtain dial-up access to a TCP/IP Internet.

- Modem (Network)

Uses an underlying serial driver. Drives a modem by passing it the necessary Hayes AT command.

- FAT filesystem (filesystem)

Uses an underlying block driver. FAT12 and FAT16 are supported. The FAT filesystem should be used in conjunction with the file name mapper driver in order to create an Elate compatible filesystem.

- SCSI disk (block)

Uses an underlying SCSI control driver.

- Elate filesystem (filesystem)

Uses an underlying block driver. The Elate filesystem is designed to be used on Flash ROM devices and on host filesystems e.g. DOS/FAT via the file block driver. The same would apply to disks and streaming realtime data.

- TCP/IP (protocol)

Uses an underlying network driver.

- Merge filesystem (filesystem)

‘Merges’ multiple disparate filesystems into a single namespace. Read/write access to a read –only filesystem can be enabled. It works by invisibly copying files from a read-only filesystem to a read/write filesystem where necessary.

- Layered mouse

Uses an underlying character driver. This driver is to support a mouse connected to a remote system, where the remote system is not running Elate. The underlying character driver is a serial driver.

- Layered pen

Uses an underlying character driver. This driver is to support a pen device connected to a remote system, where the remote system is not running Elate. The underlying character driver is a serial driver.

- Playback Driver

The playback device takes recorded pen and keyboard actions, and then plays them back so that it behaves as if it were a device driver for the physical devices that were used to record them.

- Debug Driver

This is used for tracing calls into and out of a driver. For more information please see *The Debugger User Guide.*

6.12 Filesystems Available

There are various filesystems available. The most suitable will depend on various factors such as:

- Is int**e**nt to be hosted or non-hosted?
- Is the storage mediaFlash, ROM, magnetic disks?
- Is there any read-only storage?
- Is Elate embedded into a device?

The features of the various Elate filesystems have been briefly described in the previous section.

6.13 Display Device Drivers

Elate supports two types of display:

- Text (character oriented)
- Graphics (pixel oriented)

6.13.1 Text Displays

The text display device driver offers character oriented access to a display. The API is documented in *dev/display/api.html*. The driver is accessed through open, write and close methods. A sequence of commands are supplied to the display device driver through the write method. These commands are documented in *dev/display/sequences.html*. Two sets of escape sequences are supported:

- VT52 style
- Elate style

The VT52 escape codes provide backwards compatibility with legacy Elate applications and also ports of UNIX applications. The new escape sequences are designed to be more efficient and more flexible than traditional VT52 type commands.

6.13.2 Writing a Device Driver

For more information on writing a device driver, please refer to *The Device Driver Design Guide* manual, available on request.

6.14 Character Codes in Elate

The packet returned by the keyboard device is a single 32 bit integer. The API for raw keyboard devices specifies this integer as an Elate raw keycode. These raw key codes are defined in *dev/keyboard/keyboard.inc*.

Bit 31 of this raw keycode is set if the packet indicates the release of the key. The remainder of the keycode is a valid Unicode character.

Keys that cannot be represented by Unicode characters, such as the cursor keys, have codes defined in the user space of Unicode. The Elate codes are between 0xE000 and 0xEFFF.

A keyboard device will usually make no attempt to maintain the shift-state or provide keyboard cooking, so the Unicode character will refer to the un-shifted state. A separate keyboard cooker is needed if the shifted states are required. See the standard system keyboard cooker documentation for more information (*elate/dev/keyboard/cookdev/user.html*).

The above integer format is somewhat inefficient in terms of memory usage; for this reason the code is often converted into a multi-byte character format, which is very memory efficient.

The keyboard cooker is essentially a wrapper for the class *dev/keyboard/cooked/class*, the keyboard cooker performing additional tasks such as processing only typed events and managing shift states, according to its installed cooking table.

The shell, for example, uses the keyboard cooker driver to convert integer Unicode into multi-byte encoded Unicode characters.

The intent multimedia toolkit does not use the keyboard cooker driver to convert to multi-byte format. It utilises the *dev/keyboard/cooked/class* directly in order to process shift-state and generate up/down/typed events. These events specify a 32-bit integer Unicode character plus shift- state.

The multi-byte encoding standard used in Elate is UTF8. This encoding scheme is also supported in the ANSI C library functions.

7. Kernel

7.1 Kernel Overview

The kernel is an advanced, portable real-time kernel written in VP. It uses services provided by the CII (CPU Isolation Interface), PII (Platform Isolation Interface) and certain device drivers (e.g. timer device), in order to achieve true platform independence.

This section provides a brief overview of kernel features. For more detailed information please refer to *The Reference Manual for the int**e**nt Kernel*.

7.2 Features

Some of the main features of the kernel include:

Portability, size and speed	User defined scheduling algorithms
Pre-emptive multi-tasking	Wide range of Inter Process Communication (IPC) mechanisms
Task priorities	Customisable
Dynamic binding	Heterogeneous multi-processing

The kernel provides the features needed to build small, fast, real-time embedded systems.

7.3 Kernel Services

The kernel provides the following main services:

- Tool management
- Inter-process communication
- Signals
- Exception handling
- Scheduling
- Kernel notification functions
- Mini atomic blocks
- Kernel device functions
- Static areas support
- Time functions
- Process management
- Memory management
- Timers
- Callbacks
- Atoms
- Atomic list functions
- AVL tree management functions
- Named data areas (NDAs)
- Entropy collection services

The following sections describe each of these areas in more detail.

7.3.1 Tool Management

The kernel has facilities for loading and binding tools. The kernel uses a special device driver called the ‘tool loader’ to search for tools on disk and load them. The tool loader will call the translator if the tool contains VP code, the translator returning the tool in native form.

Typically a tool is loaded when a process qcalls the specified tool. The actual mechanism depends on the particular type of *qcall* made. There are three main types of qcall (aside from embedded):

- ♦ non-virtual
- ♦ virtual
- ♦ virtual+fixup

The default *qcall* type is non-virtual. With this *qcall* all tools referenced by the calling tool are loaded (and translated if required) when the calling tool is loaded. Thus, all the tools that may be required are available in memory at the start of program execution. This mechanism provides maximum run-time speed, but is less efficient in memory use because tools that have been loaded may not actually be required, depending on execution path.

With virtual *qcall*s the referenced tool is not loaded until dictated by the run-time execution path. At this point the tool is searched for in memory and if not present the tool is loaded from disk. On completion of execution of the virtual tool the calling process decrements the tool's reference count. The reference count is a count of all processes currently referencing (using) that tool. Note only one copy of a given tool exists in memory at any one time, so if four processes were using a particular tool simultaneously it would have a reference count of four and there would not be four copies of the tool in memory. When the reference count of a tool is zero it may be flushed from memory by the kernel. The advantage of the virtual tool is that memory requirements when the calling tool is loaded are minimal, as virtual tools only get loaded when required. The disadvantage is that there is a small run-time performance hit as the requested tool is loaded and translated.

The virtual+fixup tool is similar to the virtual tool in that it is only loaded on request (if not already in memory). However, when the virtual+fixup tool completes the reference count of the tool is not decremented. This means that the virtual+fixup tool will not be flushed should the kernel decide to flush dereferenced tools when resources are low. The advantage of this is that any subsequent calls to the virtual+fixup tool will not require loading and translation of the tool. So on the first call to the virtual+fixup tool, performance is comparable to a virtual *qcall* and for subsequent calls performance is comparable to a non-virtual *qcall*. Note however that V+F calls patch the original call in the calling tool, which is why subsequent calls are almost as fast as non-virtual. Virtual calls cannot do this, so even if the tool is already in memory it has to be found on the tool list, and is therefore slower.

The kernel provides the following tool management calls. Please refer to *The Reference Manual for the int**e**nt Kernel* for more information. Note that these tools will only be used under special circumstances; the more usual approach is to load and manage tools via the *qcall* mechanism.

- *sys/kn/tool/open* – scan memory for tool, if not found tool loader is called
- *sys/kn/tool/deref* – decrement reference count
- *sys/kn/tool/ref* – increment reference count
- *sys/kn/tool/flush* – flush all unreferenced tools from memory
- *sys/kn/tool/flushname* – flush specified tool from memory
- *sys/kn/tool/lookup* – return pointer to header of tool containing this address
- *sys/kn/tool/getname* – given pointer to header of tool, name is returned in buffer
- *sys/kn/tool/stktrace* –navigate up a call stack (used to construct stack trace when Java™ exceptions are thrown).
- *sys/kn/tool/memstats* – return information about tool-list memory usage
- *sys/kn/tool/getrefcount* – return the reference count of a given tool
- *sys/kn/tool/setspngo* – set stack pointer and jump to specified address, clearing up stack
- *sys/kn/tool/enumerate* – enumerate all the elements on the tool-list

7.3.2 Process Management

A process can be thought of as a program in execution. The int**e**nt kernel supports multiple processes running in parallel. Multi-tasking is supported via timeslicing, and, on multi-processor systems, true parallel processing is also supported.

On many operating systems there is a distinction between threads and processes. On these systems threads have the following properties:

- ♦ usually faster and more memory efficient to create threads, rather than processes
- ♦ threads can share certain types of data such as file descriptors, signal tables, etc.
- ♦ threads spawned from an application usually operate within its memory space, allowing them to share data more efficiently and without memory protection

int**e**nt uses a very lightweight process based programming paradigm. The programmer does not explicitly create 'threads' per se; rather the programmer spawns new, lightweight processes. int**e**nt processes have the following properties:

- the creation of processes in int**e**nt is fast and memory efficient
- memory protection between processes is not enforced
- the sharing of system resources and data between processes is supported e.g. common file table etc

In fact, in int**e**nt a thread is thought of as a route through executable code in a sequence of tools.

An int**e**nt process can have any of the following states:

- Suspend
- Sleep and suspend
- Running
- Dormant
- Sleep
- Ready

Brief descriptions of process related kernel services are given below. Please note that *sys/kn/proc/** calls fall into two types:

- act on self (calling process)
- act on another specified process

Process creation and deletion

- *sys/kn/proc/create* – create a new process (which must then be started, as when it is created it will be in the DORMANT state)
- *sys/kn/proc/delete* – delete a process; this operates only on processes in the DORMANT state
- *sys/kn/proc/start* – start a created process, moving it into the READY state.
- *sys/kn/proc/exit* – put calling process (itself) in the DORMANT state
- *sys/kn/proc/terminate* – terminate specified process, moving it to DORMANT, unless it is in the RUN state

Process creation helper functions

- *sys/kn/proc/exec/local* – create and start a process on the same processor as the calling process
- *sys/kn/proc/exec/remote* – create and start a process on the specified processor
- *sys/kn/proc/exec/any* – create and start a process on a processor chosen by the kernel

Process control

- *sys/kn/proc/sleep* – put calling process (itself) to sleep and block until ‘woken.’
- *sys/kn/proc/wake* – wake specified process, moving it to the READY state
- *sys/kn/proc/suspend* – suspend specified process. The calling process cannot suspend itself. The suspended process is moved to the SUSPEND state or, if it was sleeping, the SLEEP&SUSPEND state.
- *sys/kn/proc/resume* – unsuspend specified process, moving it to READY or, if it were both sleeping and suspended, to SLEEP.
- *sys/kn/proc/wait* – wait for a specified child process, if terminated, and delete it
- *sys/kn/proc/chld* – wait for a child process to terminate
- *sys/kn/proc/devsleep* – put calling process to sleep until event occurs (for devices only)
- *sys/kn/proc/atexit* – add routine to list of routines to be run when process exits

Process scheduling

- *sys/kn/proc/deschedule* – give up current time slice
- *sys/kn/proc/ideschedule* – cause scheduling to occur (for use by PII only)

Process parameters

- *sys/kn/proc/setparams* – modify scheduling parameters of specified process
- *sys/kn/proc/getparams* – get scheduling parameters of specified process
- *sys/kn/proc/chpri* – change the priority of a calling process

- *sys/kn/proc/getpri* – return the priority of a calling process
- *sys/kn/proc/chppid* – set the parent process PID to a new value

Spawning

- *sys/kn/proc/spawn/make* – create a spawn structure to be used by *sys/kn/proc/create*
- *sys/kn/proc/spawn/emake* – POSIX style version *of sys/kn/proc/spawn/make*
- *sys/kn/proc/spawn/modfd* – modifies file descriptor in spawn structure, adding one if there is already one present
- *sys/kn/proc/spawn/modglobs* – modifies global data initialisation in spawn structure, adding one if there is already one present
- *sys/kn/proc/spawn/modparent* – modifies SPAWN_PARENT record in spawn structure, adding one if there is already one present
- *sys/kn/proc/spawn/modstack* – modifies SPAWN_STACK record in spawn structure, adding one if there is already one present
- *sys/kn/proc/spawn/modsig* – modifies SPAWN_SIGMASK in spawn structure, adding one if there is already one present
- *sys/kn/proc/spawn/modstklimit* - add SPAWN_STACKLIMIT record to spawn structure

7.3.3 Scheduling

The kernel supports the following scheduling algorithms:

- Rate Monotonic (RM)
- Maximum Urgency First (MUF)
- Deadline Monotonic Scheduling (DMS)
- Minimum Laxity First (MLF)
- Earliest deadline First (EDF)
- Fixed Priority Scheduling (FDS)

It is also possible to utilise several scheduling policies at the same time by assigning different policies to different task priority ranges. A process can have a priority in the range 0 – 255, where 0 is the highest priority.

7.3.4 Process ID (PID)

int**e**nt defines a portable interface to the process table data structure. There are two types available:

- ♦ Fixed size
- ♦ Hierarchical

The fixed size model is suitable for small, embedded systems, where the number of processes is limited and known at the outset. The hierarchical process table is more suitable for systems where the requests of downloaded applications are unknown, where there are a large number of processes or where the actual number of processes is otherwise unpredictable. Note that customer definable process tables are made possible by implementing the process table access tools in the *sys/kn/proc/pid* directory. In the case of remote access the hierarchical table is required.

The default int**e**nt process table is the fixed size process table. For more information please refer to *The Reference Manual for the int**e**nt Kernel.*

int**e**nt also supports two types of PID:

- ♦ Local
- ♦ Network

Normally local PIDs are used but where a number of int**e**nt kernels are on a network and need their processes to communicate network unique PIDs must be assigned to processes. This is done by the kernels agreeing on which unique numbers can be allocated so that a conflict in network PIDs does not occur.

The programmer should be aware that it is possible to spawn remote processes from a process that has a local PID. However, signals from the child process may not pass back to the parent process as there is no network PID for it.

7.3.5 Inter Process Communication (IPC)

The intent kernel supports several process synchronisation mechanisms. These include:

- ♦ Counting semaphores
- ♦ Event Flags
- ♦ Reader / Writer Locks
- ♦ Mutexes
- ♦ Event Tools

These can be used individually, or to construct IPC mechanisms such as mailboxes (described below).

7.3.6 Counting Semaphores

Semaphores provide access protection for a resource, to which a limited number of simultaneous accesses are permitted. A semaphore maintains a count of the number of vacancies available for each resource. Semaphore operations include:

- *sys/kn/sem/init* – initialise a semaphore
- *sys/kn/sem/destroy* – destroy an unnamed semaphore
- *sys/kn/sem/trywait* – wait on a semaphore, non-blocking
- *sys/kn/sem/trymwait* – decrement the semaphore count by the given amount without blocking
- *sys/kn/sem/wait* – wait on a semaphore
- *sys/kn/sem/timedwait* – wait on a semaphore with timeout
- *sys/kn/sem/post* - post to a semaphore
- *sys/kn/sem/getvalue* - get the value of a semaphore

7.3.7 Mutexes

A mutex is a mechanism for providing mutual exclusion of access to a device, data etc. A mutex can be used to ensure that only one process has access to a particular resource at a time. Mutex operations include:

- *sys/kn/mtx/init* – initialise mutex
- *sys/kn/mtx/destroy* – destroy mutex
- *sys/kn/mtx/lock* – lock mutex, blocking
- *sys/kn/mtx/trylock* – lock mutex, non-blocking
- *sys/kn/mtx/timedlock* – lock mutex with timeout
- *sys/kn/mtx/unlock* – unlock mutex
- *sys/kn/mtx/islocked* – return lock status of mutex
- *sys/kn/mtx/siglock* - locks a mutex, retrying if interrupted by a signal
- *sys/kn/mtx/sigtimedlock* - attempt to lock a mutex with timeout, retrying if interrupted by a signal

7.3.8 Event Flags

Event flags allow multiple processes to wait for a combination of up to 32 specified events. Event flag operations include:

- *sys/kn/evf/init* – initialise an event flag structure
- *sys/kn/evf/destroy* – destroy an event flag
- *sys/kn/evf/set* – set the flag pattern of an event flag
- *sys/kn/evf/clr* – clear the flag pattern of an event flag
- *sys/kn/evf/wait* – wait on event flag until specific condition occurs
- *sys/kn/evf/trywait* – test event flag for specified event flag pattern, non-blocking
- *sys/kn/evf/timedwait* – wait on event flag for specific condition, with timeout
- *sys/kn/evf/info* – get event flag information

7.3.9 Event Tools

Event tools provide a generic means of creating synchronisation objects, via a system of event trackers. Each event tracker has 32 bits of internal state, the semantics of which vary, e.g. semaphores and event flag tools are based on event tools. Event tool operations include:

- *sys/kn/event/init* – initialise an event tracker
- *sys/kn/event/destroy* – destroy an event handler
- *sys/kn/event/wait* – wait on an event tracker, blocking, no timeout
- *sys/kn/event/trywait* – wait on an event tracker, non-blocking
- *sys/kn/event/timedwait* – wait on an event tracker, blocking with timeout
- *sys/kn/event/alter* – alter the state of an event tracker
- *sys/kn/event/info* – get the value of an event tracker
- *sys/kn/event/wait_fn* – wait on an event tracker in the specified way, blocking, no timeout
- *sys/kn/event/trywait_fn* – as above, non-blocking with timeout
- *sys/kn/event/timedwait_fn* – as above, blocking with timeout
- *sys/kn/event/alter_fn* – alter the state of an event tracker in the specified manner
- *sys/kn/event/enumerate* – call a specified function for each caller waiting on an event tracker

7.3.10 Reader/Writer Locks

Reader/Writer locks provide a locking mechanism that allows access of different kinds to resource it protects. Multiple processes may access the resource in one mode, if they do not wish to modify it. Modifying data can only be done in the other mode and is limited to one process at a time. Reader/Writer lock tools include:

- *sys/kn/rwlock/init* – initialise a r/w lock
- *sys/kn/rwlock/destroy* – destroy an r/w lock
- *sys/kn/rwlock/wait* – wait on a r/w lock, blocking, no timeout.
- *sys/kn/rwlock/trywait* – wait on a r/w lock, non-blocking
- *sys/kn/rwlock/timedwait* – wait on a r/w lock, blocking with timeout
- *sys/kn/rwlock/unlock* – unlock a r/w lock
- *sys/kn/rwlock/upgrade* – when holding lock as reader, become a writer

Note that ‘waiting’ here refers to such occurrences as waiting until the lock is gained or until the event in question has happened, and so on. Blocking refers to waiting until the designated action has occurred while blocking from doing anything else. Non-blocking means that after an initial attempt is made no further action will be taken. Timeout simply means that attempts will cease after a certain time period.

7.3.11 Mailboxes

Mailboxes provide asynchronous message passing between processes, and can also be used to implement synchronous message passing. Mailbox operations include:

- *sys/kn/mbox/alloc* – allocate a mailbox
- *sys/kn/mbox/free* – free a mailbox
- *sys/kn/mbox/send* – send message to mailbox
- *sys/kn/mbox/read* – read mail from mailbox
- *sys/kn/mbox/tryread* – read mail from mailbox, non-blocking
- *sys/kn/mbox/timedread* – read mail from a mailbox, with blocking and timeout
- *sys/kn/mbox/setcallback* – set the callback function for the specified mailbox
- *sys/kn/mbox/enumerate* – call the specified function with the message list of the mailbox

7.3.12 Synchronisation Groups

Extensive facilities are available for collecting process synchronisation objects together into a collective structure referred to as a ‘synchronisation group.’ Semaphores, mutexes, event flags and mailboxes can all be added to a specific group. This makes it easier to write applications that need to wait for one of several events to occur. Available synchronisation groups include:

- *sys/kn/sgrp/init* - initialises a synchronisation group
- *sys/kn/sgrp/destroy* - destroys a synchronisation group
- *sys/kn/sgrp/mtx_assoc* - associates a mutex with a synchronisation group.
- *sys/kn/sgrp/sem_assoc* - associates a semaphore with a synchronisation group.
- *sys/kn/sgrp/mbox_assoc* - associates a mailbox with a synchronisation group.
- *sys/kn/sgrp/evf_assoc* - associates an event flag with a synchronisation group
- *sys/kn/sgrp/sgp_assoc* - associates a synchronisation group with a synchronisation group
- *sys/kn/sgrp/mtx_disassoc* - disassociates a mutex from a synchronisation group
- *sys/kn/sgrp/sem_disassoc* - disassociates a semaphore from a synchronisation group.
- *sys/kn/sgrp/mbox_disassoc* - disassociates a mailbox from a synchronisation group
- *sys/kn/sgrp/evf_disassoc* - disassociates an event flag condition from a synchronisation group.
- *sys/kn/sgrp/evf_destroy* - isassociates all event flag conditions for a specified event flag from a *synchronisation group*
- *sys/kn/sgrp/sgp_disassoc* - disassociates a synchronisation group from another synchronisation group.
- *sys/kn/sgrp/wait* - waits on a synchronisation group (blocking, no timeout)
- *sys/kn/sgrp/trywait* - waits on a synchronisation group (non-blocking)
- *sys/kn/sgrp/timedwait* -waits on a synchronisation group (blocking, with timeout).

7.3.13 Memory Management

int**e**nt uses an object based memory allocation scheme. Examples of available memory objects include:

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

These memory objects could be mapped to different memory allocator instances, or simply be mapped to a single allocator instance.

Various allocator classes are also available:

- *sys/kn/mem/debug* – debugging allocator
- *sys/kn/mem/pii* – call underlying host OS to allocate and free
- *sys/kn/mem/gwc* – ‘good worst case’ allocator, default intent allocator
- *sys/kn/mem/bp* – ‘bomb-proof’ allocator, resists all common forms of corruption

The details of underlying memory objects and allocator objects are hidden from the application programmer. The programmer uses the following tools to allocate and free memory:

- *sys/kn/mem/allocstk* – allocate memory for process stacks
- *sys/kn/mem/allocdata* – allocate memory for data
- *sys/kn/mem/allocsys* – allocate data memory for the system itself
- *sys/kn/mem/alloccode* – allocate memory to store dynamically loaded tools
- *sys/kn/mem/allocmail* – allocate memory to be sent as mail messages
- *sys/kn/mem/allocdef* – allocate data memory which needs to be shared by a number of processes and which must exist after the allocating process terminates

Other useful tools include:

- *sys/kn/mem/free* – free up specified memory block
- *sys/kn/mem/allocaligned* - allocates memory with a specified alignment

- *sys/kn/mem/realloc* – reallocate a specified memory block
- *sys/kn/mem/check* – check structure of all system memory objects
- *sys/kn/mem/lookup* – get pointer to named memory object
- *sys/kn/mem/size* – return size of block of memory

7.3.14 Timer Management

intent provides two types of timer. The first, periodic, 'times out' after an initial period, and will then continue to expire repeatedly after the specified period. The second, monoshot, expires once after the specified time period.

On timeout/expiry the action may be:

- wake a specified process
- send a signal to a specified process
- call a timer handler

The timer functions include:

- *sys/kn/timer/set* – set up a timer using the priority of the calling process
- *sys/kn/timer/dset* – set up a timer using the priority specified in the timer data structure
- *sys/kn/timer/unset* – unset timer

7.3.15 Interrupt Handling

The following restrictions apply to interrupt handlers. Of these, the first three restrictions apply to the PII rather than the kernel.

- Stack – interrupt handlers execute with a stack setup by the PII. This stack is of fixed size. Overrunning the stack will cause a system crash.
- Register usage – interrupt handlers should preserve all native registers.
- Memory usage - memory accessed by the interrupt handler should be locked using *sys/kn/mem/lock*. This applies to both code and data memory.

See the release documentation on the PII for further detail on interrupt handlers.

- Code usage – The following kernel tools may only be called from an interrupt handler or timer handler:

- *sys/kn/int/evf/set*
- *sys/kn/int/proc/wake*
- *sys/kn/int/proc/terminate*
- *sys/kn/int/proc/resume*
- *sys/kn/int/callback/set*
- *sys/kn/int/reslock/unblockall*
- *sys/kn/int/event/alter*
- *sys/kn/int/sig/kill*
- *sys/kn/int/mbox/send*
- *sys/kn/int/proc/suspend*
- *sys/kn/int/proc/setparams*
- *sys/kn/proc/getparams*
- *sys/kn/int/reslock/unblock*
- *sys/kn/int/sem/post*
- *sys/kn/int/event/alter_fn*

7.3.16 Exception Handling

When an exception occurs, the First Level Exception Handler (FLEH) pushes the exception details onto the stack in the form of an EXC structure. It then invokes the Second Level Exception Handler (SLEH), which proceeds to call a number of Third Level Exception Handlers (TLEHs) in sequence, until the exception is successfully processed. 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. There can be at most one System Debugger TLEH in the system, and no more than one Process Debugger TLEH per process.

TLEHs operate in an Exception context under the following restrictions.

- A TLEH may not perform a long-jump, or call any other tool that performs a long-jump.
- A TLEH may not remove itself from the list of handlers.
- When it returns, a TLEH must return a code indicating the action the SLEH should take next.
- TLEHs may not call any other tools.

The kernel provides the following exception handling functions.

- *sys/kn/exc/set* - register a process's exception handler with the kernel
- *sys/kn/exc/unset* - deregister a process's exception handler with the kernel
- *sys/kn/exc/setsys* - register a system-wide exception handler with the kernel
- *sys/kn/exc/unsetsys* - deregister a process's exception handler with the kernel
- *sys/kn/exc/setdbg* - register a debugger’s exception handler with the kernel
- *sys/kn/exc/unsetdbg* - deregister a debugger’s exception handler with the kernel
- *sys/kn/exc/throw* – throw a software exception

7.3.17 Signals

POSIX 1 signals are supported. POSIX 4 signals are not currently supported.

Signals are generated due to one of the following categories of events:

- Hardware exceptions – illegal instructions, invalid memory references, arithmetic exceptions etc.

- Abnormal software conditions – writing to a pipe with no readers, termination of a child process, attempts to virtually call tools that do not exist etc.

- Program initiated events – expiration of an alarm, IPC using SIGUSRn signals, termination of signals etc.

- User-initiated events - the user pressing control-C or control-Y at the keyboard, running the kill program, etc.

There are three possible actions for a signal:

- SIG_DFL - perform default action for signal
- SIG_IGN – ignore the signal
- Pointer to signal handling function – on delivery of the signal the receiving process calls the signal handler indicated by a pointer. The handler will normally return to the receiving process, but may be made to return to a location previously specified by *setjmp* by using the *longjump* tool.

Many systems use the concept of a ‘supervisor or privileged mode’ and a ‘user mode’. In these systems signals can only be delivered to a process while it is in ‘user mode’. This gives protection from signal delivery while sensitive operations such as I/O or manipulation of system data structures are being performed. int**e**nt has no concept of supervisor mode or user mode, however it can provide the same level of protection as supervisor mode by switching off signal delivery to a process under certain circumstances. Note that this does not affect a process’s signal mask or any pending signals. Most critical regions within 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 a mutex. Signals are effectively re-enabled when the process unlocks the mutex. This removes the danger that, for example, a signal handler may *longjmp* out of a critical region. Use of the MTX_SIGMASK feature is of particular interest to programmers of software that would normally be thought of as running in supervisor mode e.g. device drivers.

The currently available signal functions include:

- *sys/kn/sig/kill* – send a signal to a process
- *sys/kn/sig/action* – examine or change signal action
- *sys/kn/sig/procmask* – examine or change blocked signals
- *sys/kn/sig/pending* – examine pending signals
- *sys/kn/sig/raised* – return set of signals which have been raised but not yet taken
- *sys/kn/sig/suspend* – wait for a signal
- *sys/kn/sig/setflag* – disable or enable signal handling
- *sys/kn/sig/signal* – set handler for specific signal

For manipulating sets of signals:

- *sys/kn/sig/emptyset* – create an empty set
- *sys/kn/sig/fillset* – create a full set
- *sys/kn/sig/addset* – add a signal to a set
- *sys/kn/sig/delset* – delete a signal from the set
- *sys/kn/sig/ismember* – test to see if signal is a member
- *sys/kn/sig/setpending* – modify the set of pending signals, replacing it with those specified
- *sys/kn/sig/orpending* – modify the set of pending signals, to include those specified

7.3.18 Callbacks

A callback is a function or tool that can be called from another process, but operates in the context of the process that ‘owns’ the callback. An example of the use of callbacks is in asynchronous device drivers. When a lengthy I/O task completes a callback to a handler function can be invoked, this can then indicate that the operation is complete and any further required operations can now be carried out. This prevents the device driver from having to enter a wasteful polling loop or sleeping and thus blocking the process that calls the driver.

Callback facilities include:

- *sys/kn/callback/set* – set a callback
- *sys/kn/callback/unset* – unset a callback
- *sys/kn/callback/setflag* – disable/enable callbacks
- *sys/kn/callback/process* – process any pending callbacks
- *sys/kn/callback/occurred* – returns the value of the PF_CALLBACK_OCCURRED flag
- *sys/kn/callback/clr_occurred* – clears the PF_CALLBACK_OCCURRED flag
- *sys/kn/callback/pending* – show whether a callback is pending for the caller

7.3.19 Named Data Areas (ND As)

The basic concept behind the NDA is that the intent kernel allows a pointer to a data area to be associated with a string or ‘name’. This has the advantage that a number of processes running on a chip can access a data area without having to know a specific pointer. This is very useful for implementing processor wide data structures that may be shared by a number of processes.

The NDA services are:

- *sys/kn/nda/name* – associate the specified pointer with the specified string
- *sys/kn/nda/del* – delete the NDA record for the specified string
- *sys/kn/nda/find* – lookup the NDA record for the specified string

7.3.20 Atoms

Atoms provide a facility whereby strings can be represented more efficiently as a unique integer. The kernel provides facilities for creating, deleting and looking up atoms. Each processor in an int**e**nt system possesses an atom table, which records the mappings between a string and its associated integer.

The int**e**nt kernel stores atoms in different ways, depending on whether they are created statically with the sysgen utility, or dynamically by the kernel itself.

Static atoms last for the lifetime of a system and are in the system image (which can be in RAM or ROM), so that they cannot be modified. They have no reference count and do not cease to exist if unused by the system. Static atoms need less memory than dynamic atoms and can be compressed to a greater degree without a large run-time penalty.

Dynamic atoms are either stored in a singly linked list. Currently, the default is the linked list, but the other can be chosen at 'sysgen time.' Each dynamic atom maintains a reference count, and may be deleted if this falls to zero. Adding an atom that already exists will increment its reference count.

The user does not need to know whether an atom is static or dynamic, and can call the functions described below on any atom, although calls to deref, delete etc, will have no effect on a static atom. Atom functions available include:

- *sys/kn/atom/add* – return atom value for specified string, create a new atom if required
- *sys/kn/atom/del* – dereference the specified atom, delete if unreferenced
- *sys/kn/atom/find* – return the atom value corresponding to the specified string
- *sys/kn/atom/getname* – return a copy of the string corresponding to the specified atom
- *sys/kn/atom/ref* – increment the reference count of an atom

7.3.21 Kernel Notification Functions

Within int**e**nt, applications, device drivers, debuggers and so forth are able to subscribe to various system events so as to be notified when they occur. This notification takes place in the form of a call to a function registered by the subscriber.

The following kernel notification functions are provided.

sys/kn/notify/announce - announce that an event has taken place
sys/kn/notify/subscribe – register to be notified when a certain event occurs
sys/kn/notify/subscribe_name – register to be notified when a certain named event occurs
sys/kn/notify/unsubscribe – withdraw subscription to be notified when a certain event occurs
sys/kn/notify/generator – register as a generator of a particular kind of event
*sys/kn/notify/generator*_name – register as a generator of a particular kind of named event
sys/kn/notify/ungenerator – cease to be a generator of a particular kind of event

7.3.22 Linked List Functions, Atomic

Linked list operations are sensitive to being interrupted in a multitasking environment. It is possible for a list to be left in an inconsistent state. This could easily cause a system crash where the lists manage important system information. For this reason the kernel provides atomic list manipulation functions.

The functions provided are:

- *sys/kn/atomic/addhead* – add a node to the head of the list
- *sys/kn/atomic/addtail* – add a node to the tail of the list
- *sys/kn/atomic/addnode* – add a node after the specified list node
- *sys/kn/atomic/addnodeb* – add a node before the specified list node
- *sys/kn/atomic/removehead* – remove the node from the head of the specified list
- *sys/kn/atomic/removenode* – remove the specified node from its list
- *sys/kn/atomic/movelist* – move the entire contents of one list onto another

7.3.23 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.

The following MAB management functions are provided:

- *sys/kn/mab/begin_s* – enter a scheduler MAB
- *sys/kn/mab/end_s* – leave a scheduler MAB
- *sys/kn/mab/begin_t* – enter a thread MAB
- *sys/kn/mab/end_t* – leave a thread MAB
- *sys/kn/mab/begin_i* – enter an interrupt MAB
- *sys/kn/mab/end_i* – leave an interrupt MAB

7.3.24 AVL Tree Management

AVL trees are a particular 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. 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.

AVL trees allow searching, inserting and removing operations on a tree of size n to be carried out in log(n) time. The AVL tree is ordered and a node which is ‘less’ in the chosen metric than another is considered to be left of it. Thus the ‘greatest’ node in the tree is the rightmost mode, and so on.

The kernel supplies the following AVL tree management functions:

- *sys/kn/avl/init* - initialises an AVL tree header
- *sys/kn/avl/init_rs* - initialises an AVL tree header with relocatable stack
- *sys/kn/avl/init_dup* - initialises an AVL tree header to permit elements with duplicate keys
- *sys/kn/avl/init_dup_rs* - initialises an AVL tree header with relocatable stack to permit elements with duplicate keys
- *sys/kn/avl/move_rs* – switch the location of the relocatable stack
- *sys/kn/avl/deinit* - deinitialises an AVL tree header
- *sys/kn/avl/insert* – inserts a new node into an AVL tree
- *sys/kn/avl/remove* – removes a node from an AVL tree, rebalancing the tree
- *sys/kn/avl/removeall* – removes all the nodes from an AVL tree
- *sys/kn/avl/removerange* - removes elements within the given range from an AVL tree
- *sys/kn/avl/find* – finds an element within an AVL tree
- *sys/kn/avl/findkey* - finds a node with a specified key value within a tree
- *sys/kn/avl/maximum* –returns the maximal element within an AVL tree
- *sys/kn/avl/minimum* – returns the minimal element within an AVL tree
- *sys/kn/avl/enumerate* – enumerates all the elements within an AVL tree, calling a function for each
- *sys/kn/avl/size* – returns the number of elements within an AVL tree
- *sys/kn/avl/check* – checks the consistency of an AVL tree
- *sys/kn/avl/ubound* – finds the leftmost element greater than or equal to a key
- *sys/kn/avl/lbound* – finds the rightmost element less than or equal to a key
- *sys/kn/avl/walkleft* – finds the next element left from a given element
- *sys/kn/avl/walkright* – finds the next element right from a given element

7.3.25 Kernel Device Functions

In intent, 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:

- device instance pointer (least significant 32-bits)
- the processor on which the device is situated (most significant 32-bits)

The use of the *devstart* command has already been discussed in the device driver section of this manual.

The flags are 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 processor.

Device functions include:

- *sys/kn/dev/lookup* – look up a device in the system mount table*
- *sys/kn/dev/rlookup* – look up a device in the system mount table*
- *sys/kn/dev/mount* – add a device to the system mount table
- *sys/kn/dev/unmount* – remove the specified device from the system mount table
- *sys/kn/dev/mount_delayed* – adds a delayed-mount record for a device into the system mount table
- *sys/kn/dev/start* – add a new device driver to running system
- *sys/kn/dev/stop* – unmount and stop a device

**Lookup* takes a name and returns a device, while *Rlookup* (R for 'reverse') takes a device pointer and returns the name of that device.

7.3.26 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 provided by the kernel dedicated to the rapid retrieval of the addresses of static areas, and is not responsible for the allocation and deallocation of static areas, their format or their contents.

A 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.

The kernel includes the following static areas support functions.

- *sys/kn/statics/statics_get* – find the data associated with a cache, or allocate it if necessary
- *sys/kn/statics/statics_delete* – delete the statics associated with the key and delete the cache

There are also the following functions in the statics configuration block:

- *alloc* – allocates a static area
- *new* – performs initialisation on an automatically allocated static area
- *init* – if setup length is 0 or more then the space is automatically allocated
- *wait* – wait for statics area to be initialised
- *deinit* – deinitialise the statics area
- *delete* **-** deletes the statics area by freeing it

7.3.27 Kernel Entropy Collector

The purpose of the entropy collector is to provide a source of high quality randomness. This randomness can then be used for applications that require randomness, such as cryptographic key generation utilities. The entropy in the system is ‘trapped’ by the device drivers in the system and then pooled by the kernel into an entropy reservoir. Requests for entropy from applications can only be serviced by the kernel if there is sufficient entropy in the entropy reservoir or entropy data pool.

In cases where entropy is not required or is not supported by underlying hardware, the kernel entropy collector can be replaced by a null version. At system build time (sysgen time), it can be specified whether the ‘real’ entropy collector is to be used or not. If not a group of stub routines are used, which support the device driver entropy API but actually perform no-ops.

Entropy collector functions include:

sys/kn/entropy/add – add entropy to the collector data pool
sys/kn/entropy/add_time – add timer entropy to the pool
sys/kn/entropy/reg-time – register the timer clock resolution
sys/kn/entropy/get – extract entropy from collector pool
sys/kn/entropy/getrand – extract entropy from collector pool
sys/kn/entropy/get_async – extract entropy asynchronously
sys/kn/entropy/get_abort – cancel asynchronous entropy request

7.3.28 Kernel Time Functions

sys/kn/time/get – get the kernel time

8. The Sysbuild Utility

The sysbuild utility is the primary image creation program for intent. It operates by taking a sysbuild file, and then calling upon the underlying functionality of Sysgen to generate an image file. A sysbuild file describes the application in terms of the tools and data files required, as well as how the program should be run. An image file is a bootable intent image which can be downloaded to the target hardware.

To generate a bootable system image from an intent command line, the following should be used:

```
sysbuild <platform> <appsysfile>
```

Here, <*platform*> specifies the platform for which the image is being created.

<*appsysfile*> specifies the application’s sysbuild file, that is to say the .sys file from which the image file is to be generated. All application sys files must have the extension *.sys*, but it does not need to be specified on the command line.

Any option supported by *sysgen* is also supported by *sysbuild*, including the following:

-*m* (-*nom*)
    When this option is used, *sysbuild* generates a map file, containing a list of the components written to the output image file, and their locations. This option is enabled by default but may be disabled using the –*nom* option.

-*n* (-*non*)
    When this option is used, VIRTUAL+FIXUP *qcalls* are changed into normal *qcalls* This option is enabled by default but may be disabled using the –*non* option.

-*s* (-*nos*)
    Strip resource names from tools where possible. This option is enabled by default but may be disabled using the –*nos* option.

int**e**nt supports the creation of application system configuration files and platform system configuration files for use with *sysbuild*.

An application sys file is divided into the following stages:

- **SETUP** - Sets up the configuration required for the specific application. In the case of a PJava application, for example, WANT_PJAVA will need to be defined so that the PJava libraries are available.
- **DEPENDS** - Sets up optional parts of the application's configuration depending on the platform, a process which is only occasionally required.
- **ROMTOOLS** - Lists the tools needed to run the application. There will usually be two lists, a list of executables and a list of data files.
- **RAMTOOLS** - Contains a list of tools that must be stored in the platform's read write memory area. This section is usually unnecessary.
- **APPS** - Contains the information required to start the application.

A platform sys file is comprised of the following stages:

- **SETUP** - Sets up the configuration required for the specific platform. This may include setting some AVE parameters, or defining the serial port configuration for the Sejin keyboard etc.
- **DEPENDS** - Sets up any optional parts of the platform configuration depending on the requirements of the applications and other areas.
- **MEMORY** - Defines the memory layout for the platform, including the platform boot tools, memory regions etc.

Further platform system configuration information is contained within *devices.sys*, a file that defines the devices supported by the platform. This list consists of platform specific device drivers and generic Elate device drivers.

The platform system configuration information may also include a platform specific script, sys/platform/<platform>/sysbuild. If this file exists, it is run once the image has been created. This can be used to transform the image before downloading, through byte swapping, Srecord creation, etc.