The Elate® Device Driver Design Guide

Version 1.31

1. Introduction

Elate®'s unique translation technology enables applications to be transparently transferred across different hardware platforms. Programs are written for a Virtual Processor (VP) allowing independence from the underlying technology. The VP Binary code generated is translated into the target processor native code at load time and is executed as native code at runtime. Source programs can be written in the assembler language for the Virtual Processor (VP Code), by writing in 'C' or 'C++' or by writing native code for the target processor.

1.1 What are 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 piece of software, 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.

1.2 The Architecture of an Elate Device Driver

The Elate operating system supports an object based programming style. A Device Driver within Elate is programmed as an object. Therefore, it is strongly recommended that a full appreciation of object oriented programming within Elate is acquired before attempting to write a Device Driver. This information is available in the manual, '*Object Programming with Elate*'.

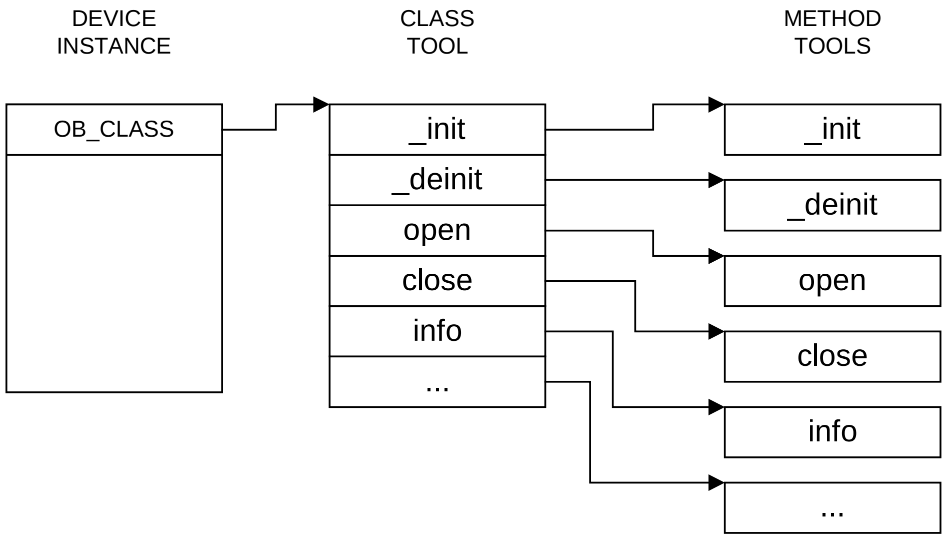

Figure 1 Architecture of a Device Driver

1.2.1 Which Programming Language?

Currently, there are restrictions as to which language is available to code each section of a Device Driver. The table below outlines which language can be used for each section.

Language	App. Interface	Class Code	Method Code	ISR Code
VP	Y	Y	Y	Y
C	Y	N	Y✘	Y
C++	Y	N	Y✘	Y
Native	Y	N	Y	Y

Table 1 Language Availability

✘ - It is not possible to program the method code using native, ‘C’ or ‘C++’ if it is coded within the main body of the class code. However, if the method makes a *qcall* to an external tool, the source code to this external tool can have been written in VP, C or C++ (See the manuals “*VP Tool Programming Guide*” and “*Elate Tool Programming Guide (C)*”).

1.2.2 Loading a Device Driver

For an application to use a Device Driver, it must be mounted. Elate provides various ways of easily doing this (see Chapter 6). 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.

All applications, whether originally written in ‘C’, ‘C++’, VP, or native code, can access the device by using the <*name*>, which it has been given when placed in the Mount List. 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.

For those programmers intending to write in C or C++, please refer to the manual “*Elate Tool Programming Guide (C).*”

1.2.3 Remote Devices

Elate technology also has the capability of allowing an application on one processor to utilise a Device Driver on a different processor. This is of great importance, as Elate is capable of allowing any number of heterogeneous processors in a network to work together, in parallel. (See Remote Device Driver Access, Chapter 7).

1.3 Preparing to Write a Device Driver

Before undertaking the implementation of a Device Driver with Elate, knowledge of the device and an understanding of its characteristics and ambiguities is required. The device will have special features that will affect the driver’s implementation and how those features are supported. The device manual itself is therefore essential.

At the back of this manual is a Device Driver template with an actual implementation of a specific device. Studying this will help in understanding the implementation of Device Drivers for Elate.

1.3.1 Memory Mapping (Platform Isolation Interface (PII))

The Platform Isolation Interface (PII) provides portability of some kernel tools by separating the independent sections of the kernel from those that are platform dependent.

Memory mapping between physical and virtual addresses is done by PII *map* functions. Device Driver programmers will require information about mapping. All information on PII functions, such as *map*, and other platform dependencies can be found in "*The Platform Isolation Interface Reference Manual,*" contained within the Elate build, within the *sys/pii/* directory.

1.3.2 I/O Access

It is possible to access the hardware using input and output assembler macros, which perform I/O through a system specific I/O space.

- *ioin* Reads a unit of data from I/O address space
- *ioout* Writes a unit of data to I/O address space
- *ioinblk* Reads a block of data from I/O address space
- *iooutblk* Writes a block of data to I/O address space

1.3.3 Exclusive Software Resource Access

When an application requests a service from a Device Driver, it is quite often the case that the device will only allow one process at any one time to have access to the variables or hardware, in order to ensure data integrity. An exclusive access technique to ensure the integrity of any operations, whilst performing a service, is therefore required.

Elate kernel mutex objects can be used to guarantee exclusive access to critical sections of code, memory, I/O ports, etc.

1.3.4 Exclusive Hardware Resource Access (I/O or Memory)

On initialisation it may be advisable to lock the device (using *dev/lockio*), defined by its 'base address,' 'length' and 'type.' If the base address and length does not overlap with any of the previously locked items (of the same type), then the new item is considered to be locked.

1.3.5 Protecting a Memory Area against Paging Out

During interrupt routines and Direct Memory Access (DMA), a technique for protecting memory regions containing code or data from paging out is required, to ensure data integrity. The Elate mechanism for locking pages is *sys/kn/mem/lock.*

1.3.6 Read and Write Policy

When creating a Device Driver which is capable of reading or writing (or both), the nature of the read/write policy will affect the performance and memory requirements of the Device Driver.

Three forms of reading and writing to a device are possible possible - Blocking, Nonblocking and Asynchronous. Device drivers targeted at specific applications need only provide the method required by that application. Normally device drivers should provide all three forms.

However, in most cases, the device driver only needs to provide the asynchronous forms (for example, *reada* and *writea*). The base class for all device drivers will implement the synchronous forms (for example, *read* and *write*) by calling the asynchronous forms followed either by a wait operation (blocking) or a cancel operation (nonblocking).

If this is likely to cause a problem for the target applications, the device driver should also provide its own asynchronous forms.

- **Blocking** – the driver blocks until the IO is complete.

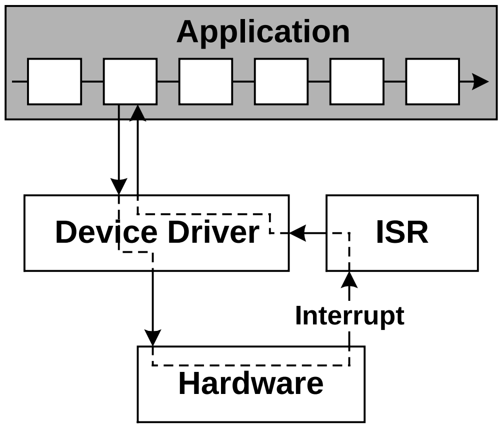

Figure 2: A Blocking Call

- **Non-blocking –** the driver returns immediately having processed some data (if it is able to).

- **Asynchronous –** the driver returns immediately having initiated the IO. At a later point it will indicate that the IO has been completed.

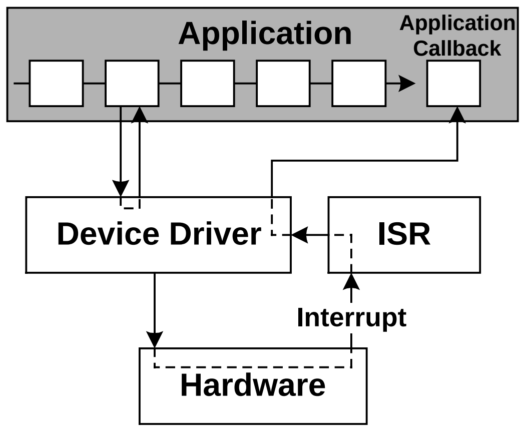

Figure 3: An Asynchronous Call

2. Creating a Device Driver Object in Memory

In order for an application to call a service from a device, it is necessary for an instance of the Device Driver object to be in memory. An allocation and de-allocation policy must be decided upon by the Device Driver programmer but the conventional way of allocating memory for the instance is by creating an allocator tool called *_new*. A tool carries out memory allocation because the object has not yet been created and a *method* cannot be called from an object that does not yet exist!

Once the instance data structure has been successfully allocated, the instance header is initialised to point to the class tool. The rest of the memory will have undefined data within it which will also require initialising. This is performed by the *_init* method. (See Chapter 3.1)

The instance data structure must be large enough to contain the private instance variables, with space for the standard memory block header of size MH_SIZE. Size is always object specific and is usually defined within an include file. The memory is allocated by using the Elate function *sys/kn/mem/allocdef*. Since there is no guarantee that the C Library is available, it is not possible to use the C Library function for this.

The assembled *_new* tool must be placed in the same directory as the Device Driver's class, i.e. *<pathname>/_new*, where *<pathname>* is the same as the class directory.

Example Code:

```
tool 'dev/*/_new'
      ent - : p0
      cpy MH_SIZE + DEVICE_SIZE,i0
      qcall sys/kn/mem/allocdef (i0: p0 i~)
```

After successful memory allocation, an instance pointer to the memory block header, defined by MH_SIZE, is returned. The instance pointer must be adjusted to point to the start of the un-initialised memory area that is beyond the memory block header. The instance pointer in VP is always the first pointer available.

```
      if p0!=0
           add MH_SIZE,p0
```

To initialise the object header and reference the class, the macro *refclass* is used with the full name of the class tool. *refclass* searches for the class tool, bringing it into memory from store, and translating it only if it is not already present on the tool list.

```
            refclass p0,dev/*/class
      endif
      ret

toolend
```

If the *_new* tool is successful, OB_CLASS is set-up to point to the instance of the class tool containing the methods. If the allocation fails the return value is 0 (OB_SIZE is defined in the include file *lang/asm/include/equs.inc*).

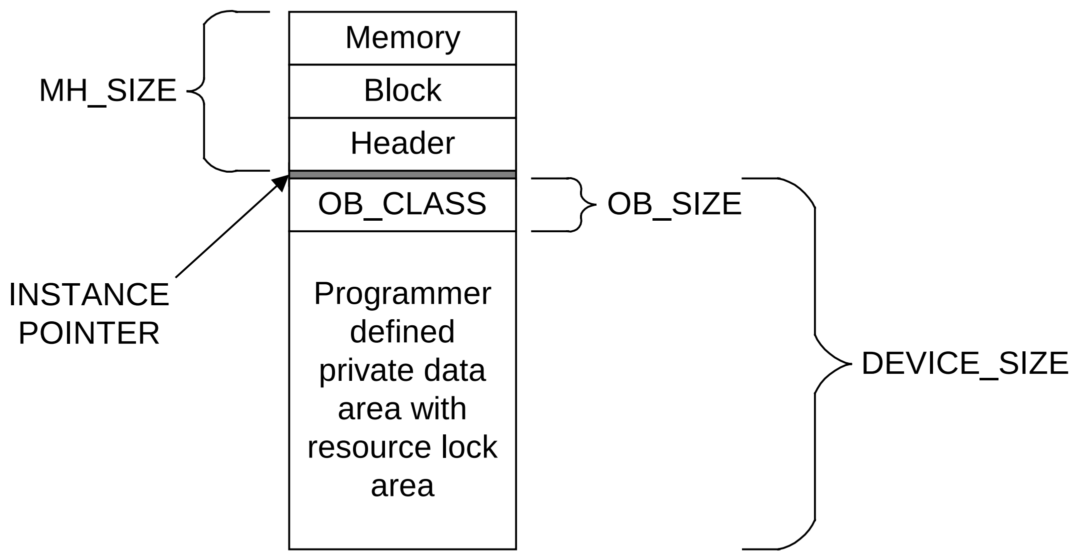

Figure 4 Schematic of Instance Data Structure in memory

3. Defining a Class and Method Coding

A Device Driver within Elate is an object and, like other objects, can inherit from a parent class. If a new Device Driver belongs to an existing family of devices, it inherits from the existing parent driver thereby gaining access to common methods (services). An example of this is *dev/mouse/pcbus*, where the pcbus mouse driver inherits common mouse methods coded in *dev/mouse*. For the implementation of a new family of Device Drivers, the *dev/<newfamily>* driver must inherit from the generic base class called *dev/class*.

dev/class allows the inheritance of File Descriptors (FD) from remote systems, performing a reverse look-up in the mount table to ensure that the correct instance of the device driver is being used. *dev/class* not only allows FD inheritance but it also implements the following:

• method *open*	opens a device from an application
• method *close*	de-references the handle for the object and closes the device
• method *reference*	increments the reference count for the specified handle
• method *getflags*	returns the flags which are currently set
• method *setflags*	changes flags currently set, for blocking/unblocking etc.

It is possible to re-implement any of the above methods in a device driver. However, it must be noted that if one is changed, all must be changed.

dev/<newfamily> must include the code associated with Remote Devices (See Chapter 7).

3.1.1 Class code

The class code is the framework within which the object's methods are coded. *class* is a special type of tool provided by Elate which has specific macros provided to build the class contents. The name of the class is defined immediately after the first macro *class*. The name is the full <*pathname*> plus class, e.g. *class 'dev/*/class'*. To inherit from another device driver class, the full <*pathname*> plus class, e.g. *class dev/*/class*, omitting the quote marks.

```
class 'dev/mouse/pcbus/class',dev/mouse/class,VP
```

The *classend* macro defines the end of the class. All the methods for the object's services are coded within these two macros.

3.1.2 Include files

It is not unusual, when programming in VP, for the *_new* tool to be coded in the same source file. Therefore the include file which defines the memory size must be included here. These are entered before the class code. The standard Tao include file, 'tao inc,' must also be included. It is only necessary to enter include files once per source file. (See Manual – "*VP Tool Programming*").

3.1.3 Method code

A *method* defines a service which a Device Driver object can perform. Each method manipulates the hardware by using the variables within the instance data structure plus parameters from the application, to provide the service and possibly return a result. Coding a Device Driver method is identical to coding any other object method, using the macro *method* and terminating with *ret*.

There are a number of methods that must be defined for a Device Driver. These are *_init*, *_deinit*, *info, reference, close* and *open* as well as *defaultmethod.* The *_init* method initialises the object instance while the *_deinit* method reverses it. The *info, reference, close, open* and any other methods are private to the particular device. Coding a method can be done in two ways:

- By putting all the code within the *method* macro (currently this means it must be written in VP).
- By making a call to an external tool (the tool could be written in 'VP,' 'C', 'C++' or native).

Conventionally, if a tool is coded in native, a second VP (or C, or C++) tool is always coded to ensure system portability. An advantage of making an external call to a tool is that it allows dynamic downloading and translation from storage only when required. By adding VIRTUAL+FIXUP after the name of the tool the tool will be loaded only on the first occasion the call is made, and from that point on is fixed in memory.

Example Code:

```
.include ‘tao’
.include '/dev/*/device.inc'
class ‘dev/*/class’,VP

     method _init
     ent p0 p1 i0:i0
          …
          ret

     method _deinit
     ent p0:i0
          …
          ret

     method other
     ent p0 p1 i0 i1:i0
          qcall dev/*/other,(*:*),VIRTUAL+FIXUP
          ret

     defaultmethod
          entd
          ret
classend
```

Note:

One implementation of a method may be to set up the hardware to allow it to generate an interrupt. To avoid wasting CPU cycles whilst waiting for the hardware to complete the command, the process should suspend. Setting up the hardware and the ISR are device specific but possible implementations may involve the use of the *sys/kn/proc/sleep* and *sys/kn/mbx/timedread* functions, with the ISR resuming the suspended process by use of the *sys/kn/int/proc/wake* function.

3.2 Initialising the object in memory

Once the *_new* tool has allocated the instance data structure, it must be initialised with the private or constant data for the object, including any mutex structures if resource locking is required. All of this is device specific but some of the considerations and procedures are outlined below.

The *_init* method is called after the *_new* tool has been successfully called. This may be carried out automatically by using the Elate provided utilities (see Chapter 6) or directly by the application itself. Any default values set up by the *_init* method may be overwritten at device initialisation, as user configured arguments can be processed from the command line.

3.2.1 Locking the I/O address

Once the instance data structure is initialised, the Device Driver may need to request permission for exclusive access to a range of I/O space. An atomic test and set can be carried out by making a call to the tool *dev/lockio.* If the range is free it is reserved. The task is not de-scheduled or suspended on failure.

3.2.2 Detecting and setting up hardware

The implementation of detecting and setting up the hardware is device specific, but if the hardware detected is not the one expected then the Device Driver must fail initialisation. However, if the hardware detected is correct, it must be set up for use by the Device Driver. Part of the set up must be to ensure that no interrupts are generated before the ISR is ready for use. Any pending interrupts must therefore be cleared.

3.2.3 Accessing shared memory

A Device Driver may utilise a shared memory area. In this case, the variables are shared between the Device Driver method code and the ISR and can be modified by either. This memory area may be obtained from system memory, pre-allocated as part of the device instance data structure, or through a private allocation technique specific to the actual hardware. If the latter, it must then be initialised to hold the variables that are to be used by the ISR.

3.2.4 Loading the Interrupt Service Routine (ISR)

When the hardware is correctly set up, the ISR is loaded with any necessary locking. The IRQ number and interrupt routine are set up for the interrupt number given, as well as the address of the loaded code contained in the ISR tool. An Elate tool, called *dev/loadisr*, is provided to do this.

3.2.5 Generating Interrupts

For hardware that generates interrupts, any previous interrupt sources must be cleared. Interrupts can then be enabled, but as an interrupt could be generated immediately, the ISR must make provision for this. Implementation is hardware specific.

4. The Interrupt Service Routine (ISR)

The ISR is code, which services the interrupts generated by the hardware. How the ISR services the interrupt is device specific and cannot be addressed here.

However, care should be taken if the device is sharing the same IRQ number with any other devices. The ISR must check that the hardware which generated the interrupt is the device's own hardware. If the Device Driver has ascertained that it is not the correct hardware, the ISR should exit with a *ret* and Elate will call the next chained interrupt handler.

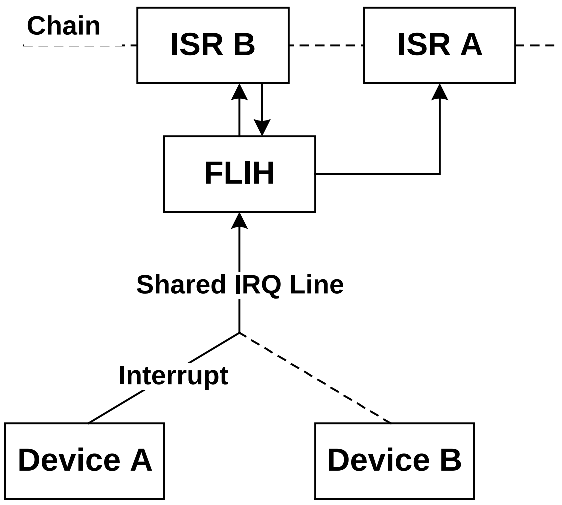

Figure 5: Sharing IRQ Numbers

If the interrupt is generated by the hardware being maintained, then the ISR must service it. After the interrupt has been serviced, any waiting Device Driver processes should be woken up.

Example code:

```
.include 'tao'

tool 'dev/*/isr'

      ;Inputs: p0=Data pointer specified to loadisr
      ;Outputs: None

      ent p0:-

      cpy.p MY_STATUS,p1
      i0in.6 p1,i0
      bool i0!?DATA_BIT,exitisr
      cpy.p MY_DATA,p1
      i0in.6 p1,i0
      cpy.b i0,[p0+SAVED_DATA]
      cpy [p0+PID],i0
      qcall sys/kn/int/proc/wake,(i0:i~)
exitisr:
      ret
toolend
```

Only certain tools can be called from the ISR. Generally it is wisest to assume that a tool cannot be called from within an ISR unless the documentation for that particular tool states that this is permitted. For API and more detailed information on each specific tool, please refer to the manual '*The Elate Kernel*' or use the on-line help within the Elate shell.

- *sys/kn/int/proc/wake*
- *sys/kn/int/proc/suspend*
- *sys/kn/int/proc/resume*
- *sys/kn/int/proc/terminate*
- *sys/kn/int/proc/getparams*
- *sys/kn/int/proc/setparams*
- *sys/kn/int/sem/post*
- *sys/kn/int/evf/set*
- *sys/kn/int/mbox/send*
- *sys/kn/int/timer/set*
- *sys/kn/int/timer/unset*

Interrupt Service Routines can be coded in C or C++ in the normal way. Please refer to the manual '*Elate Tool Programming Guide (C)*' before attempting to write in C or C++.

5. Calling a device from an application

For an application to be able to access the services provided by a device, the Device Driver object must be in memory and have been initialised.

The application must obtain the Device Driver object's instance pointer and then open the device. Once this has been successfully achieved, the application can make method calls to the object for the services it requires. Once finished with, the application must close the Device Driver object, so that it is available for another application to gain access.

Applications can be written in 'C', 'C++' or VP code. VP code has many 'C' library function tools already available. Figure 3 gives the algorithm used for opening and calling a method from a device. On the left of the diagram are the 'C' code calls and on the right is the VP code. The 'C' calls are all available in VP as library calls.

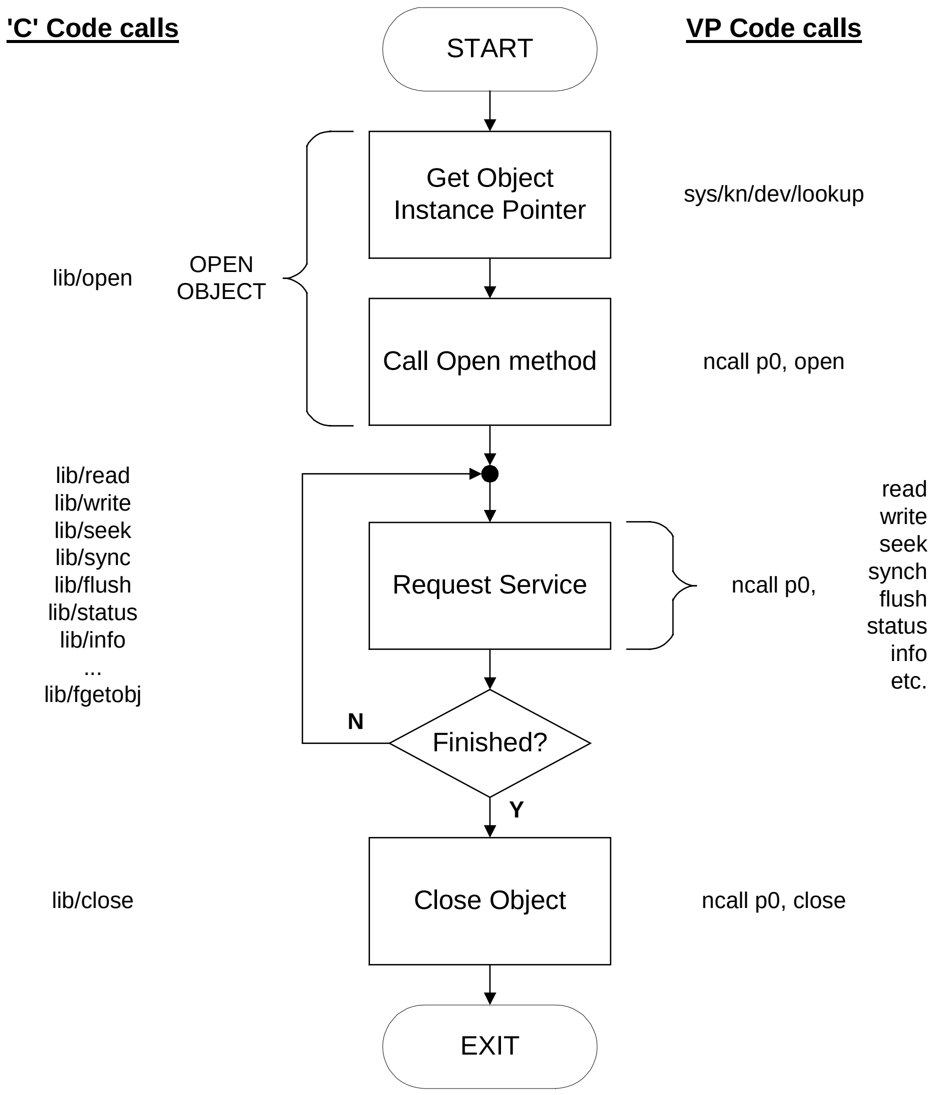

Figure 6 Algorithm of application code for calling a method (service)

5.1.1 Looking up a Device

The kernel function *sys/kn/dev/lookup* looks up the specified device by matching the specified name against mounted devices. It is common for matches to be incomplete: for example, a lookup of the name *var/tmp/tmpfile0001* may match with the *var* device. In this case, a string pointer would be returned, pointing to the end of the matched string within the input string, eg *tmp/tmpfile0001*. Note that the leading slash is stripped off. Obviously this string represents the filename relative to the root of the matched device.

Many systems have a filesystem device mounted as an empty string. This will match any name. In the above example, however, the lookup would still match with *var* as that is a closer match.

5.1.2 Tool *lib/fgetobj*

The tool *lib/fgetobj* returns the instance data and handle associated with a file descriptor, thereby enabling an application to ncall a device object directly. Its API is as follows:

Inputs

- Ptr = location where the instance data will be stored
- Ptr = location where the handle will be stored
- Int = file descriptor

Outputs

- Int = handle, or -1 if error (errno set to error code)

Note:

By using the ‘C’ Library function *lib/open*, the file descriptor also becomes inheritable, whereas an ncall to a method *open* does not.

6. Loading a Device Driver into Memory

For an application to be able to make calls to a device, an instance of the class of the Device Driver for that device must be in memory. There are two ways provided for easily creating a Device Driver instance in memory and these are:

- *.obj* as used by the Sysgen utility
- *devstart*

Sysgen provides a mechanism for building a single file containing all the kernel functions and system boot code required. It may also contain an application; in addition, all its dependent tools required to run that application. This file can either be loaded into memory from a file system or blown into ROM. For more detailed information please refer to *'The Sysgen Reference Manual'.*

Both *devstart* and the *.obj* call the *_new* tool for the specified Device Driver class. If the call to the *_new* tool is successful, the class tool for the Device Driver is now loaded into memory, if not already available.

If a method was written with a call to an external tool, it may or may not be loaded until the method is actually called from the application, at which time it would be loaded transparently.

Once the instance data structure is in memory the *_init* method is called. Any parameters entered on the command line are now processed. Pointers to the command line parameters are automatically passed into the *_init* method for processing.

If the call to the method *_init* is successful, the Device Driver's <*name*> is loaded into the *Mount List* making the device available for use by applications. in order to gain access to the Device Driver, applications must specify <*name*> exactly as in the Mount List.

The <*name*>, which is used within the Mount List, is specified by the user when either *devstart* or *.obj* is run. By using different names, more than one instance of a Device Driver can be in memory at any one time. The <*name*> within the Mount List is associated with a pointer to the Device Driver's class instance data variables.

6.1.1 How to use *devstart*

Type in the command directly from a keyboard or use the configuration file. *devstart* takes the first parameter to be the <*name*> by which the Device Driver is to be known in the Mount List. The second parameter is taken to be the <*pathname*> of the directory location of the Device Driver class. Any additional parameters are passed to the *_init* method.

Example code:

```
$devstart /device/joystick dev/joystick/pc -p$201
```

This code loads a PC specific joystick driver into the Mount List with the name */device/joystick*. The -p specifies the port address which is the default address for the joystick interface for the PC.

6.1.2 How to use *.obj*

The syntax for loading a Device Driver using *.obj* is as follows:

```
.obj (newtool="<_new filename>", mountstr="<mountname>", cmdline="<textual
parameters>")
```

Example code:

```
.obj (newtool="dev/joystick/pc/_new", mountstr="/device/joystick",
cmdline="joystick -p$201"
```

If any of the operations are unsuccessful, *devstart* and *.obj* will back out cleanly, de-allocating the memory and returning without having loaded the Device Driver.

6.2 Unloading a Device Driver

An Elate shell command is provided called *devstop* which removes a Device Driver from the system, thereby making it no longer available for use by applications*.*

devstop searches the Mount List for the <*name*> specified. If the <*name*> is found, it calls the *_deinit* method of the Device Driver. The *_deinit* method is called to de-initialise the hardware and the *devstop* deletes the object instance. The <*name*> is removed from the Mount List. This is a matched pair with *devstart.*

Example code:

```
$devstop /device/joystick -@0
```

6.3 Device Driver Version Information

The *info* method coded in the class code will hold information such as the version number of the Device Driver as well as other information about the device. To be able to look at this information, an Elate shell command is made available called *devinfo*. The Mount List is searched for the name specified and if found, calls the *info* method which will print the returned information to *stdout*. For more information upon this command, please see the "*Shell User Guide.*"

Example code:

```
$devinfo /device/joystick
```

6.4 API Specifications for methods

All device drivers implement certain methods: *_init, _deinit, info, open, close, reference, flush* and *sync*. The open method may return ENOTSUP, in which case the *close*, *reference* and (for Character, Pointer and Keyboard device drivers) *getflags* and *setflags* methods are not required. Some of these may be handled by the base device driver class (see *defaultmethod*) if the default action is suitable for the device. All other methods are family specific.

Methods normally return a success or error indicator. Error codes are always in the range -128 to -1, but should be tested using the Elate VP2 macros *iferrno*, *ifnoterrno*, *boolerrno*, *boolnoterrno*, *breakiferrno*, *breakifnoterrno* or the C macro *is Elate errno*. All other values indicate success.

The following table shows the methods that are implemented by all device drivers of a particular family.

Method	**Block**	**Character**	**Pointer**	**Keyboard**	**Filesystem**
_alias	*	*	*	*	
_deinit	*	*	*	*	*
_getattr					*
_init	*	*	*	*	*
_setattr					
bcleara	*				
bsize	*				
cancel	*	*	*	*	
clearerror	*				
close	*	*	*	*	*
defaultmethod	*	*	*	*	*
fgetattr					*
flush	*	*	*	*	*
fsetattr					*
fsync	*				*
getattr					*
getflags	*	*	*	*	

info	*	*	*	*	*
mkboot					*
mkdr					*
open	*	*	*	*	*
read		*	*	*	*
reada	*	*	*	*	
reference	*	*	*	*	*
remove					*
rename					*
seek					*
setattr					*
setflags	*	*	*	*	
statfs					*
status		*	*	*	
statusa		*	*	*	
sync	*	*	*	*	*
write		*	*	*	*
writea	*	*	*	*	

Please note that *read* and *write* need not be implemented if the driver implements *reada* and *writea*. The base class for drivers will read and write to *reada* and *writea*.

Other methods may also be implemented. These methods would be device driver specific. For example, *setbaud* would be implemented by a serial driver. Other examples of device driver specific methods are:

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

Please note that block device drivers do not support non blocking reads or writes.

6.4.1 Method _*init*

Initialises the object data structure.

Inputs:

- p0 = Instance pointer
- p1 = argv
- i0 = argc

Outputs:

- i0 = success, else error code

Errors codes:

- EACCES = Permission denied obtaining resource
- EINVAL = Invalid parameter
- ENODEV = No such device exists
- ENOMEM = Not enough memory

Description

This method initialises the object data structure, set up by the *_new* tool, allocating any memory required for internal buffers, ISR, list headers, etc.

6.4.2 Method _deinit

De-initialises the object just prior to the its deletion.

Inputs:

- p0 = Instance pointer

Outputs:

- i0 > 0 if success, = 0 if device still in use, else error code

Errors codes:

- EIO = Error de-initialising device.

Description

This method is called to de-initialise the object, just before the object is deleted. This includes de-initialising and restoring hardware settings, freeing any allocated memory (except the object instance data structure), closing any child processes, etc.

The return value from this method indicates whether the deinit has successfully completed, failed, or whether the device is still in use.

If the device is in use, then the hardware should not be de-initialised, a value of 0 is to be returned. The caller should not delete the object instance. The caller may retry the *_deinit* call a short while later.

If the device is not in use and the de-initialisation of the hardware is successful then a value of greater than zero is returned to the caller. The caller can then proceed to delete the object instance.

If the device is not actually in use, but there is some failure during the de-initialisation of the hardware, which prevents the device from being un-installed, then an error code is returned. This is to notify the caller that there is a failure and so prevent the caller from retrying indefinitely.

6.4.3 Method *info*

Provides information about the device object.

Inputs:

- p0 = Instance pointer
- p1 = Handle or NULL
- p2 = Pointer to buffer
- i0 = Length of buffer

Outputs:

- i0 = Number of bytes written to buffer

Description

This method copies information about the device object into a buffer provided by the caller. The handle may be NULL if the information is not required.

The format of the information is as follows:

```
int32 SIZE
int32 <Device Family>
```

Followed by one or more:

```
int32 <Device Family Information Flags>
```

Followed by an optional number of:

```
int32 <Parameter Type> <Parameter1> <Parameter2>
```

Followed by:

```
int32 DD_TERMINATOR  (32 bit zero)
<NUL terminated string>
```

The *size* field is to enable an application to find out how large its buffer is to be, so as to accommodate all the information data which may be returned. If the caller makes the info call with i0 = 4, the size of the full info block is returned. The caller can use this information to allocate a buffer of exactly the correct size.

Device Family (Class) can be one of the following:

• DF_UNDEFINED	Unknown or undefined devices
• DF_BLOCK	Block devices (e.g. HDD)
• DF_CHAR	Character devices (e.g. Serial)
• DF_SOUND	Sound devices (e.g. WAV)

• DF_KEYBOARD	Keyboard devices
• DF_POINTER	Pointer devices (e.g. Mouse)
• DF_FILESYS	File System devices (e.g. FAT)
• DF_TABLET	Tablet devices (e.g. penpad)
• DF_PROTOCOL	Protocol devices (e.g. TCP/IP)
• DF_NETWORK	Network devices (e.g. Ethernet Adapters)
• DF_MESSENGER	Messenger driver
• DF_LINK	Link devices
• DF_GRAPHIC	Graphic devices (for the intent multimedia toolkit)
• DF_AVE	AVE devices (for the intent multimedia toolkit)
• DF_TIMER	RTC timer devices
• DF_JOYSTICK	Joystick device
• DF_CPLOADER	Co-processor loader device
• DF_CPDISPATCHER	Co-processor dispatcher device
• DF_PLANE	Plane device
• DF_MSGPIPE	Message-based pipe device
• DF_CLIPBOARD	Clipboard device
• DF_POWER	Power management device

- **Device Family Information Flags:**

These define attributes of the device family. There is one common flag, DFI_MOREFLAG, which if clear indicates that this is the last flag field. If set it indicates that there are more Device Family Information Flag fields. The last Device Family Information Flag field will have the DFI_MOREFLAG flag clear.

- **Device Family Information Flags for Block devices:**

• DFI_REMOVABLE	Media is removable
• DFI_LOCKABLE	Media may be locked
• DFI_EJECT	Media may be software ejected
• DFI_MUSIC	Start, Stop and other CD Music commands supported

- **Device Family Information Flags for Character devices:**

• DFI_SERIAL	Serial device supporting setbaud
• DFI_SCATTER	Scatter/gather IO supported
• DFI_I2C	i2c device

- **Device Family Information Flags for Sound devices:**

None

- **Raw Keyboard Devices**

• DFI_REPEAT	Keyboard auto-repeats

- **Device Family Information Flags for Pointer devices:**

• DFI_2BUTTON	Two buttons available
• DFI_3BUTTON	Three buttons available
• DFI_WHEEL	Pointer has a wheel

- **Device Family Information Flags for File System devices:**

• DFI_MERGE	Merge Filesystem

Device Family Information Flags for Protocol devices:

• DFI_GETXBYY	Device supports getXbyY methods

- **For any type of device the optional fields must be one of the following:**

• DD_TERMINATOR	end marker
• DD_POLLING	1 word field
• DD_IRQ	2 word field
• DD_DMA	2 word field
• DD_MEM	2 word field
• DD_IO	2 word field
• DD_MEMRANGE	3 word field
• DD_IORANGE	3 word field

- **Example Code:**

For a Serial device loaded onto COM1 ($3F8 to $3ff inclusive), using IRQ 4, the following info block may be returned:

```
dc.i 59       ;number of bytes including length and final 0
dc.i DF_CHAR
dc.i DFI_SERIAL|DFI_SCATTER
dc.i DD_IORANGE,$3f8,$3ff
dc.i DD_IRQ,4
dc.i DD_TERMINATOR
dc.b "PC Serial Port Driver Version 2.00",0
```

6.4.4 Method *open*

Opens the device from an application.

Inputs:

- p0 = Instance pointer
- p1 = Name pointer
- i0 = Flags
- i1 = Mode

Outputs:

- p0 = Handle if success else error code

Errors codes:

- EACCES = Permission denied
- EBUSY = The object is in use
- ENOENT = The object does not exist
- ENODEV = The device does not support open

Description

This method is a request from the application to open the device. If successful a handle which refers to that open is returned. The handle is used by other methods to refer to this device in future calls.

The handle returned by this function may be any value except -128 to -1, as long as it uniquely refers to the opened device. If the device does not support opens it must return error code ENODEV.

The handle is new, and therefore is not shared with any other process in the system. The current access position should be set to the beginning of the device where appropriate.

The file status and access modes of the new handle are set according to the value of the flags parameters passed in an int. The value of this parameter is the bitwise inclusive OR of values from the lists overleaf (defined in "*lang/asm/include/filesys.inc*" and "*lang/cc/include/fcntl.h*").

Open flags
Exactly one of the following flags should be specified:

• O_RDONLY	Open for reading only
• O_WRONLY	Open for writing only
• O_RDWR	Open for reading and writing

Any combination of the remaining flags may be specified:

• O_APPEND	If this flag is set, the file offset is set to the end of the device prior to each write
• O_CREAT	If the specified file exists, this flag has no effect except as noted in the section on O_EXCL, below. If the file does not exist, it is created. In this case, the creation mode flags are significant, as it represents the file creation mode.
• O_TRUNC	If the file exists and is a normal file, and the file is successfully opened O_RDWR or O_WRONLY, it is truncated to 0 length.
• O_EXCL	Request for exclusive access to the device. This flag has no effect if the O_CREAT flag is not also set. If O_CREAT and O_EXCL are set, a check is performed for the existence of the specified file. The open action fails if the specified file already exists.
• O_NONBLOCKING	If this flag is set, operations on this file complete immediately, and do not block until the whole operation can be completed.
• O_ TEXT	This flag signifies that the file is text. Some host file systems store text files in a different format to binary files.

Any combination of the following flags may be specified:

S_IRUSR	00400 user has read permission
S_IWUSR	00200 user has write permission
S_IXUSR	00100 user has execute permission
S_IRWXG	00070 group has read, write and execute permission
S_IRGRP	00040 group has read permission
S_IWGRP	00020 group has write permission
S_IXGRP	00010 group has execute permission
S_IRWXO	00007 others have read, write and execute permission
S_IROTH	00004 others have read permission
S_IWOTH	00002 others have write permission
S_IXOTH	00001 others have execute permission

S_IAUSR	=	S_IRUSR	+	S_IWUSR	+	S_IXUSR
S_IAGRP	=	S_IRGRP	+	S_IWGRP	+	S_IXGRP
S_IAOTH	=	S_IROTH	+	S_IWOTH	+	S_IXOTH

If a device driver does not include an *open* method, the default action is to open the device and return a handle that can be used on subsequent methods.

6.4.5 Method *close*

De-references the handle for the object and closes the device.

Inputs:
- p0 = instance pointer
- p1 = handle

Outputs:
- i0 = success, else error code

Errors code:
- EBADF = The handle is invalid

Description

The close method de-references the handle. If the reference count for the handle reaches 0, the handle should be deleted and the device closed. Any outstanding asynchronous operations for the handle are cancelled.

The method should block until all the data buffered for output has been written.

Upon successful completion, a value of 0 is returned, otherwise an error code to indicate the error condition must be returned.

6.4.6 Method *reference*

Increments the reference count for the specified handle.

Inputs:

- p0 = instance pointer
- p1 = handle

Outputs:

- i0 if success, or error code if failed

Errors codes:

- EBADF = The handle is invalid

This method increments the reference count for the specified handle. The reference count is initialised to 1 when the handle is allocated by the *open* method, and it is decremented when the handle is closed.

Usage of this method allows the implementation of handle inheritance and handle duplication. If successful, the *reference* method returns any valid success value, otherwise an error code to indicate the error condition.

6.4.7 Method *flush*

Inputs:

- p0 = Object data structure
- p1 = Handle

Outputs:

- i0 = 0 if success, else error code

Error codes:

- EBADF = The handle is invalid.

This method discards all unwritten data for the specified file.

6.4.8 Method *sync*

Inputs:

- p0 = Object data structure
- p1 = Handle

Outputs:

- i0 = 0 if success, else error code

Error codes:

- EBADF = The handle is invalid.

This method ensures that all information for the specified file, including filesystem information, is written to the underlying device.

6.4.9 method *defaultmethod*

Inputs:

- None

Outputs:

- None

In most Device Drivers the following code is used:

Example Code:

```
defaultmethod
      entd
      parentclass
      ret
```

Note:
The *ent* directive must be replaced with *entd* which indicates that this is a defaultmethod and that responsibility is passed back to the parentclass of the object.

7. Remote Devices

Elate is a heterogeneous multi-processor operating environment. It is therefore useful to have a mechanism to allow an application on one processor to have access to a Device Driver on a different processor. This allows one processor to concentrate on handling the Device Drivers whilst leaving any others to handle the application, e.g. a graphics card with a specialist graphics processor.

Elate provides such a mechanism and an *alias* object is created on the same processor as an application, when this application is not on the same processor as the Device Driver. The current implementation is to have this mechanism in a base class for the family of devices from which all members of that family can inherit. It is not essential that it is done this way, but obviously it avoids extra method coding in the specific family member.

When an application makes a call to *sys/kn/dev/lookup* to open the Device Driver, the Mount List is checked for the <*name*> of the Device Driver. One of the Mount List components is a processor id number and this is also checked.

If the application and Device Driver have two different processor id numbers, *sys/kn/dev/lookup* automatically initiates the creation of an *agent process* on the processor handling the Device Driver. An *alias object* is then created on the same processor as the application calling the device. The *alias object* is able to receive *ncall*s from the application as if it were the actual Device Driver. It packages the parameters into a *mail message* sending it to the *agent process,* which in turn passes on the parameters to the Device Driver. Return parameters from the device are then re-packaged in a return message and sent back to the *alias object*, which returns these back to the application. The *agent process* and *alias object* are only created the first time the remote Device Driver is accessed. The application believes that it is talking directly to the device, regardless of whether the device is actually on the same processor, or not.

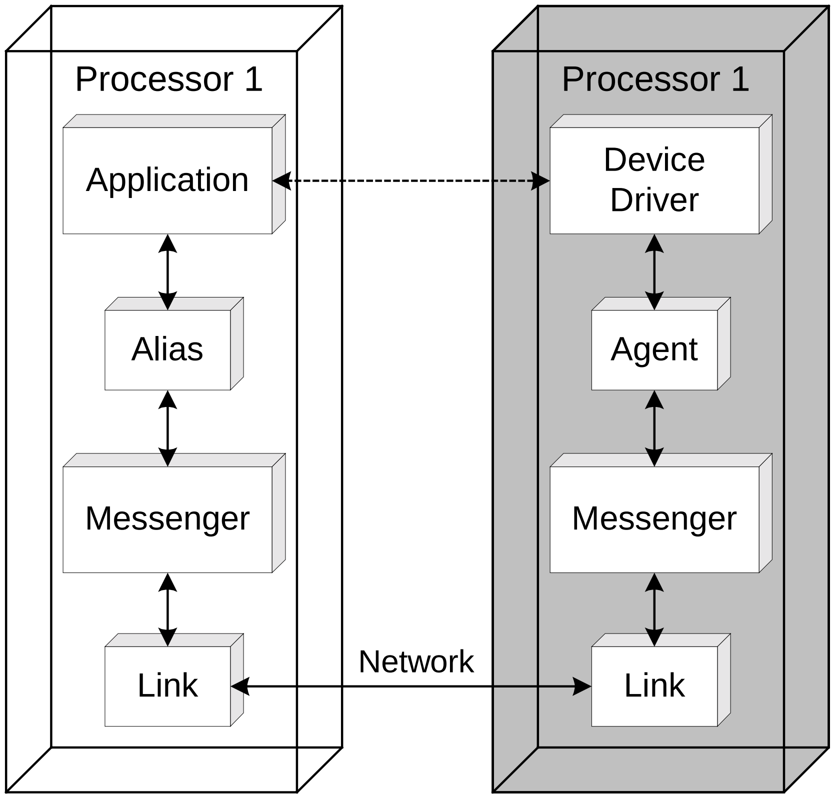

Figure 7: Remote Devices

8. Asynchronous IO Tools

Asynchronous IO (AIO) enables layers of device drivers to communicate efficiently with each other without the need for multiple helper processes for each blocked action. This is particularly important for low usage APIs, such as power management.

It should be noted that each AIO operation requires its own AIO block. This can be reused only when the AIO operation has completed.

8.1 Structure AIO

The asynchronous IO structure contains the following fields that may be used by device drivers implementing asynchronous operations:

- **List Header**

The driver can use the Elate list header at the start of the structure to queue the operation. Once the driver calls *dev/iocomplete* the List Header field may not be used.

- **AIO_OFFSET**

This long field is normally used to allow an application to specify the seek offset for the associated asynchronous read or write operation, if the device is seekable. It may be used by the device driver for any purpose, including the return of a long value.

- **AIO_LENGTH**

The driver may store any value in the AIO_LENGTH field during the operation. It is conventionally used for the length requested by the caller.

- **AIO_HANDLE**

The driver should normally save the handle in the AIO_HANDLE field when an asynchronous operation is started. This enables the driver to cancel the operation if the handle is closed.

- **AIO_CANCEL**

The driver can use this field to install a cancel function for the asynchronous operation. The field is initially set to NULL by the IO prepare tools. The driver can put the address of a cancel function in this field. If the cancel method is invoked, the device driver base class will call this function with the same parameters as passed to the cancel method.

- **AIO_DEVCALLBACK**

The driver can use this field to get a call back at process time to enable tidy up actions for an asynchronous operation. It is normally used to unlock user buffers and the AIO structure itself. The field is initially set to NULL by the IO prepare tools. The driver must ensure the field is NULL when dev/iocomplete is called if the callback is not required.

- **AIO_DEVDATABACK**

The driver can use this field for any purpose during the asynchronous operation. The value in this field is passed to the AIO_DEVCALLBACK function.

- **AIO_DEVPTR1**

The driver can use this field for any purpose.

- **AIO_DEVPTR2**

The driver can use this field for any purpose.

An example of an asynchronous input operation is provided by the int**e**nt multimedia toolkit keyboard driver in chapter 11.

8.2 AIO Tools

dev/ioprepcallback	Prepares an asynchronous IO structure to be used with any asynchronous IO method supported by a device driver
dev/ioprepdevcallback	As for dev/ioprepcallback but where the callback should be run within the context of a device driver process rather than the current user process
dev/ioprepsignal	When the AIO completes, marks the signal as pending
dev/ioprepevf	When the AIO completes, sets the event flag
dev/iogetresp	Gets the status of the AIO operation identified by the AIO structure
dev/iowait	Sleeps until the AIO operation identified by the AIO structure is complete, then returns the final status of the operation
dev/iotimedwait	This tool sleeps until the asynchronous IO operation identified by the AIO structure is complete or the timeout given by the parameter expired.
dev/iocomplete	Completes the AIO operation

9. Device Driver Helper Tools

Helper tools provide functions that are commonly required by device drivers. They are intended to make the implementation of device drivers easier, as well as reducing code size. Some of the helper tools use an extensible standard handle structure.

While some helper tools may share the same name as the device driver base class methods, they do not directly implement these base class methods. They may also differ in the parameters specified, which allows the caller to modify the functionality.

9.1 General Helper Tools

9.1.1 Helper IO Tools

dev/lockio	Locks an item defined by its base address, length and type
dev/unlockio	Unlocks an item previously locked by *dev/lockio*
dev/loadisr	Loads an Interrupt Service Routine (ISR) on behalf of a device driver
dev/unloadisr	Unloads an ISR loaded by *dev/loadisr*

9.1.2 Asynchronous IO Tools

As asynchronous IO queues may be accessed from different threads (and from interrupts in some cases), these queues need some form of exclusion protection. Although these helper tools do not mandate a specific method, certain tools may be easier to use with particular methods.

dev/status	Returns the current status of the handle
dev/setstatus	Sets the status of the handle
dev/setstatuschange	Identical to *setstatus* but with additional parameters
dev/initstatus	Sets up the status data structure for handling AIO requests
dev/deinitstatus	Deinitialises the *statusa* support mechanism
dev/statusa	A simple implementation of *statusa* for device drivers
dev/statuscancel	A cancel handler suitable for use with the *statusa* queue
dev/cancel	A simple general purpose cancel handler
dev/ioprep	Fills in the fields of the AIO structure to ready it for use
dev/cancelqueue	Cancels all requests in an AIO queue, ignoring the AIO_CANCEL field. The queue is protected by disabling interrupts
dev/cancelqueuemtx	Identical to *dev/cancelqueue* but a mutex used to protect the queue

9.1.3 Handle Management Tools

If multiple handles are supported on a single device in a device driver that supports asynchronous requests, the device driver has to keep track of both which handle an asynchronous request is attached to (so that it can be cancelled if the handle is closed) and also the list of asynchronous requests for each device. This obviously makes it harder to manipulate the asynchronous requests.

The handle management helper tools do not mandate a method for a device driver to manage handles. However, if the handles for a particular device are stored in a simple list, *dev/open* is able to check for access conflicts. These tools rely on the handle becoming invalid once the reference count reaches 0.

dev/open	Creates a new handle and checks compatibility of flags
dev/close	Deinitialises a handle
dev/reference	Modifies the reference count
dev/getflags	Returns the device flags for the handle
dev/setflags	Sets the device flags for the handle

9.2 Serial Device Driver Helper Tools

9.2.1 General Tools

dev/serial/checkflags	Checks and updates the flags supplied to the *setbaud* method

9.2.2 Modem Status Tools

These tools provide a simple implementation of *modemstatus* and *modemstatusa* for device drivers that do not require a more sophisticated implementation.

A modem status structure is defined (with size MODEMSTATUS_SIZE) which contains a copy of the physical status of the modem and the header of a list of AIO structures representing the asynchronous requests.

dev/serial/initmodemstatus	Initialises the data area used by the other modem status tools
dev/serial/deinitmodemstatus	Deinitialises the modem status structure
dev/serial/modemstatus	Simple implementation of the *modemstatus* method
dev/serial/modemstatusa	Simple implementation of the *modemstatusa* method
dev/serial/setmodemstatus	Updates the modem status and processes AIO requests for modem status

10. Error Tracking Macros

10.1 Overview

The error tracking macro scheme exists to improve the process of dealing with the various errors that can occur in a module or system. The scheme is intended to assist a programmer with identifying the errors which can occur, and remedying them.

The macros provide an easy to use mechanism that allows the programmer to build in code to a module, as it is being written, in order to deal with errors.

The macros are defined in the 'devices' include file *'lang/asm/include/devices.inc'.*

10.2 Errors

The error tracking macro scheme provides macros to DETECT, TRACE, and TRAP (but not HANDLE) errors.

Each error is classified into one of the following 'types':

PROGRAM errors
- These are errors that occur because of a programming problem
- An example of a program error is the value of a state variable being '7' when only values '0','1' and '2' are legal values

CONFIGURATION errors
- These are errors that occur because the module or system configuration is not set up correctly
- An example of a configuration error is the modem driver detecting that no ATD dial command string has been set up in the appropriate configuration file when it has been asked to dial up a link

ENVIRONMENT errors
- These are errors that occur due to a problem with the environment
- An example of an environment error is a kernel tool returning a 'no resources currently available' error to a module
- The error values defined in *'lang/asm/include/errno.inc'* ERRNO.INC are generally environment errors (e.g. ENOMEM, EBUSY)

10.3 Error, Detect, Trace and Trap Macros

The following macros are provided to DETECT, TRACE, and TRAP errors:

- PROGRAM_ERROR DETECT condition

- PROGRAM_ERROR DETECT condition

The following macros are provided to TRACE errors:

- CONFIGURATION_ERROR TRACE condition, TRACE description format string, <TRACE arguments>
- ENVIRONMENT_ERROR TRACE description format string, <TRACE arguments>

These are described below:

10.3.1 PROGRAM_ERROR

The macro is invoked as follows:

PROGRAM_ERROR DETECT condition

DETECT condition

- This is the DETECT condition to be tested

If the DETECT condition is true, the macro will cause the 'filename' and 'line number' of the macro invocation line to be output to the trace device.

The macro TRAPS the error (i.e. generates an exception to stop the program).

This macro only has one argument to be exactly synonymous with the ASSERT macro in 'C'. No facility is provided for outputting additional trace text (other than FILE and LINE NUM) to the trace device. This means that no extra trace strings will be included in object files for those modules that have the macro expansion enabled, thus minimising object file 'bloat'.

e.g:

```
      cpy.i [InstP + AWD_CONNECTION_STATE],State
      PROGRAM_ERROR (State != AWD_ACTIVATED)
      ;State should be 'activated' at this point

trace output:
 dev/msg/elate/class.asm:1269
```

10.3.2 CONFIGURATION_ERROR

This macro is used to TRACE configuration errors.

The code that DETECTS and HANDLES configuration errors should be present before and/or after the macro invocation.

The expected normal usage is for configuration error DETECTION and HANDLING code to **always** be present in the module (or to be conditionally assembled using some scheme outside the scope of the error tracking macros).

A module containing configuration error TRACE code is expected to provide a command line option, which will allow the configuration error TRACE to be switched on and off.

The macro is invoked as follows:

CONFIGURATION_ERROR   TRACE condition, TRACE description format string, <TRACE string qualifier values>

The CONFIGURATION_ERROR macro has the following arguments:

TRACE condition

- This is the 'is configuration error trace enabled?' condition to be tested
- In general a module has a control flag in its instance data, which may be set by a command line option to enable the tracing of configuration errors (e.g. the '-d' option in AWD and PPP). This condition is tested to establish if the tracing of configuration errors is enabled
- (Note that this 'trace condition' is **not** the 'has a configuration error occurred?' failure condition, i.e. it is **not** the same as the 'DETECT condition' argument passed to the PROGRAM_ERROR macro)

- The code that DETECTS and HANDLES, configuration errors should **always** be present.

TRACE description format string
- String containing description of the error

<TRACE string qualifier values>
- Optional %d %x values for the description string

If the TRACE condition is true, the macro will cause the TRACE description string to be output. The macro does not generate an exception, and the program will continue.

e.g:

```
      ;get the dial string to send to the modem
      qcall dev/network/awd/get_call_mode,(InstP BufP BufSize : Result)
      if (Result != SUCCESS)

            CONFIGURATION_ERROR ([InstP + AWD_FLAGS] bit
BAWD_CONFIG_TRACE_ENABLED),"awd/activate: no modem dial command string set
up\n"

            ;no dial string has been set up by the user in config file
            ;handle configuration error
            ;free resources just acquired
            ;report this as a 'no resources' "environment" error to PPP
      else

      endif

Trace output:
 CONFIGURATION ERROR:  awd/activate: no modem dial command string set up

NOTE that the condition '([InstP + AWD_FLAGS] bit
BAWD_CONFIG_TRACE_ENABLED)' is a TRACE condition, not a DETECT condition.
```

10.3.3 ENVIRONMENT_ERROR

This macro is used to TRACE environment errors.

The code that DETECTS and HANDLES configuration errors should always be present before and/or after the macro invocation.

The macro is invoked as follows:

ENVIRONMENT_ERROR   TRACE description format string, <TRACE string qualifier values>

The ENVIRONMENT_ERROR macro has the following arguments:

TRACE description format string
- String containing description of the environment error

<string qualifier values>
- Optional %d %x values for the description string

The macro does not perform any condition test.
The macro will cause the 'file_name' and 'line_number' to be output to 'tracef'.
The macro will cause the ‘filename’ and ‘line number’ of the macro invocation line to be output to the trace device.
The TRACE description string will also be output, allowing any returned error values to be traced.
The macro does **not** generate an exception - the program will continue.

e.g.

```
      ;get memory for circuit cell
      qcall sys/kn/mem/allocdef,(Size : CircuitCellP Size)
      if (CircuitCellP == NULL)

            ENVIRONMENT_ERROR "mpc800_i2c/open: allocdef failed
CircuitCellP=NULL\n"

            ;no memory available for circuit cell
            ;handle environment error

            ;free any other resources just acquired

            ;return 'no resources' error to caller
            cpy.p ENOMEM,HandleP
            ret
      else

      endif

trace output:
 dev/amino/intact/i2c/class.asm:1263  mpc800_i2c/open: allocdef failed
CircuitCellP=NULL
```

10.3.4 EXPANSION CONTROL MACROS

The following macros are provided to control what macro expansions take place when the PROGRAM_ERROR, CONFIGURATION_ERROR, and ENVIRONMENT_ERROR macros are invoked in a source file.

- INCLUDE_PROGRAM_ERROR_CHECKS
- INCLUDE_CONFIGURATION_ERROR_TRACE
- INCLUDE_ENVIRONMENT_ERROR_TRACE

- EXCLUDE_PROGRAM_ERROR_CHECKS
- EXCLUDE_CONFIGURATION_ERROR_TRACE
- EXCLUDE_ENVIRONMENT_ERROR_TRACE

These macros are used at the top of a source file to select the set of error tracking macro expansions that are included when the file is assembled.

For example:

- To assemble with full error checking and trace code present, the source will include:

;enable all error checking and tracing
INCLUDE_PROGRAM_ERROR_CHECKS
INCLUDE_CONFIGURATION_ERROR_TRACE
INCLUDE_ENVIRONMENT_ERROR_TRACE

- To assemble with only CONFIGURATION and ENVIRONMENT error tracing, the source will include:

;enable configuration and environment error tracing only
EXCLUDE_PROGRAM_ERROR_CHECKS

```
INCLUDE_CONFIGURATION_ERROR_TRACE
INCLUDE_ENVIRONMENT_ERROR_TRACE
```

- To assemble without any error checking and trace code present, the source will include:

```
;exclude all error checking and trace code
EXCLUDE_PROGRAM_ERROR_CHECKS
EXCLUDE_CONFIGURATION_ERROR_TRACE
EXCLUDE_ENVIRONMENT_ERROR_TRACE
```

A nested macro definition scheme is used such that (for example) the 'INCLUDE_PROGRAM_ERROR_CHECKS' macro invocation will define the full 'PROGRAM_ERROR' macro, and the 'EXCLUDE_PROGRAM_ERROR_CHECKS' macro invocation will define a 'null macro' for the 'PROGRAM_ERROR' macro.

11. Glossary of Terms

- **Dynamic Binding**

'Just-in-time' loading of tools.

Tool

Tools are re-entrant, re-locatable, loaded & bound on demand and are executable pieces of code.

- **Ncall**

Provides encapsulation (data hiding, bundling together data and access procedures), inheritance (the ability to include previously defined attributes in your new object) and polymorphism (the ability to run code with the same name and interface on different data types).

- **Qcall**

Performs a *gos* to a tool that may or may not be in memory.

- **Message**

Objects communicate by sending messages. There are different types of message which Elate is able to operate on, such as library messages, data messages and many more (see TAO_EQUS.INC). Messages are data structures. Both messages and objects are constructed from the basic node structure, but they fulfil very different roles. Objects are active entities which may send passive messages between one another.

- **Mailbox**

Messages are sent to mailboxes. An object must have a mailbox to receive messages.

- **Virtual + Fixup**

By adding Virtual + Fixup after a qcall, it indicates that a tool should be loaded on the first occasion the call is made, and from that point onwards be fixed in memory.

12. Examples

12.1 intent Multimedia Toolkit Keyboard Driver

```
.include 'tao'
.include 'dev/ave/tao/class'
.include 'ave/avo/class'

structure
struct KEY_GLB_DATA,KB_SIZE
struct KEY_GLB_AIO,AIO_SIZE
pointer KEY_GLB_AVEDEV
pointer KEY_GLB_KEYDEV
pointer KEY_GLB_KEYHANDLE
pointer KEY_GLB_DEVNAME
int32 KEY_GLB_FLAGS
size KEY_GLB_SIZE

tool 'dev/ave/dsk/keyboard',VP,F_MAIN,16384,KEY_GLB_SIZE

        ent -:-

        defbegin 0
        defp argv,ave,app,kdev,khdl,aio,msg
        defi evt,flag,key

        ;set priority to above applications
        qcall sys/kn/proc/chpri,(64:i~)

        ;get args, initialise any instance variable defaults
        qcall lib/argcargv,(-:argv,i~)
        cpy keyboardname,[gp+KEY_GLB_DEVNAME]
        clr [gp+KEY_GLB_FLAGS]

        ;process any command line options
        qcall lib/opts,(argv,options.p,gp:i~,i~)

        ;lookup ave
        qcall sys/kn/dev/lookup,(avename.p:ave,app)
        ifnoterrno ave,true
                ;open ave
                ncall ave,open,(ave,app,0,0:app)
                ifnoterrno app,true
                        ;lookup keyboard
                        qcall sys/kn/dev/lookup,([gp+KEY_GLB_DEVNAME].p:kdev,khdl)
                        ifnoterrno kdev,true
                                ;open keyboard
                                ncall kdev,open,(kdev,khdl,O_EXCL|O_RDONLY,0:khdl)
                                ifnoterrno khdl,true
                                        ;get the AIO structure for loop
                                        cpy (gp+KEY_GLB_AIO),aio

                                        ;save device instances for callbacks later
                                        cpy ave,[gp+KEY_GLB_AVEDEV]
                                        cpy kdev,[gp+KEY_GLB_KEYDEV]
                                        cpy khdl,[gp+KEY_GLB_KEYHANDLE]

                                        ;prepare the callback block (needs to be done every
time...)
                                        qcall dev/ioprepcallback,(aio,keyboard_callback.p,gp:-)

                                        ;call asynchronous read to get KB_SIZE bytes
                                        ncall kdev,reada,(kdev,khdl,gp,aio,KB_SIZE:flag)
                                        ifnoterrno flag
                                                repeat
                                                        ;wait until callback or event wakes us up
                                                        ncall app,getevent,(app,-1.l:p~,msg,evt)

                                                        ;quit if asked
                                                        if msg!=0
                                                                ;free event
                                                                ncall ave,freeevent,(ave,msg:-)
                                                                breakif evt=EV_QUIT
                                                        endif

                                                        ;leave if there was error in callback
                                                until [gp+KEY_GLB_FLAGS] geu -128

                                                ;loop until we've cancelled
                                                repeat
                                                        ;call cancel and then sleep to enable
callback
                                                        ncall kdev,cancel,(kdev,khdl,aio:flag)
                                                        breakif flag eq EINVAL
                                                        qcall sys/kn/proc/sleep,(-1.l:i~)

                                                        ;get asynchronous response and loop again
                                                        qcall dev/iogetresp,(aio:flag)
                                                until flag!=EINPROGRESS
                                        endif

                                        ;close keyboard device
                                        ncall kdev,close,(kdev,khdl:i~)
                                endif
                        endif

                        ;close ave
                        ncall ave,close,(ave,app:i~)
                endif
        endif

        ;close io, return error code
        qcall lib/exit,(0:-)
        ret

        defendnz

keyboard_callback:
        ;inputs
        ;p0=aio
        ;p1=data

        ent p0-p1:-

        defbegin 0
        defp aio,data,ave,kdev
        defi len
        defl time

        ;get the state and jump to exit if error value (stop recursion)
        qcall dev/iogetresp,(aio:len)
        ifnoterrno len
                ;load up ave device and dispatch
                if len>0
                        if len<KB_SIZE
                                qcall sys/kn/time/get,(-:time)
                        else
                                cpy [data+KB_TIMESTAMP],time
                        endif
                        cpy [data+KEY_GLB_AVEDEV],ave
                        ncall ave,dispatchk,(ave,[data+KB_KEY],time:-)
                endif

                ;prepare the callback block (needs to be done every time...)
                qcall dev/ioprepcallback,(aio,keyboard_callback.p,data:-)

                ;get the device and app instance to make the callback
                cpy [data+KEY_GLB_KEYDEV],kdev
                ncall kdev,reada,(kdev,[data+KEY_GLB_KEYHANDLE].p,data,aio,KB_SIZE:len)
        endif

        cpy len,[data+KEY_GLB_FLAGS]
        ret

        defendnz

setd:
        ;set device name
        ent p0-p2,i0:p0,i0

        cpy p1,[p2+KEY_GLB_DEVNAME]
        add 4,p0
        ret

        data

options:
        ;options table for command line processing
        dc.i ('d'),setd
        dc.i 0

avename:
        dc.b '/device/ave/'
appname:
        dc.b 'Keyboard',0
keyboardname:
        dc.b '/device/keyraw',0

toolend

.end
```

12.2 PC BUS Mouse Driver using serial hardware interrupts

```
.include 'taort'
.include 'dev/mouse/pcbus/class'
.include 'dev/mouse/pcbus/pcbus'

.remacro tracef
.endm

class 'dev/mouse/pcbus/class',dev/mouse/class,VP

        method _init
                ent p0 p1 i0 : i0

                cpy i0, i1

                ; parent class init
                pcall p0,_init,{p0 p1 i1 : i0}
                if i0>=0
                        ;sub class init
                        qcall dev/mouse/pcbus/_init,{p0 p1 i1:i0}
                        if i0<0
                                ;sub class init failed so deinit parent classs
                                pcall p0,_deinit, {p0 : i1}
                        endif
                endif
                ret

        method _deinit
                ent p0 : i0
                qcall dev/mouse/pcbus/_deinit,{p0:i0},VIRTUAL+FIXUP
                if i0!=0
                        ;parent class deinit
                        pcall p0,_deinit,{ p0 : i0 }
                endif
                ret

        method open
                ent p0 p1 i0 i1:p0
                cpy.p p0,p2
                or O_EXCL,i0                                ;Force one opener.
                pcall p2,open,{p2 p1 i0 i1:p0}
                ifnoterrno p0
                        clr [p2+(PCBUSMS_ISR_VARS+PCBUSMSISR_XPOSN)]
                        clr [p2+(PCBUSMS_ISR_VARS+PCBUSMSISR_YPOSN)]
                endif
                ret

        method close
                ent p0 p1:i0

                pcall p0,close,(p0 p1:i0)      ;Return the open count
                if i0==0                       ;If open count is zero then cancel all AIOs
                        qcall   dev/mouse/pcbus/close,(p0 p1:i0)
                endif
                ret

        method reada
                ent p0-p3 i0:i0
                qcall   dev/mouse/pcbus/reada, {p0-p3 i0:i0}, VIRTUAL+FIXUP
                ret

        method info
                ent p0 p1 p2 i0:i0
                qcall dev/mouse/pcbus/info,{p0 p1 p2 i0:i0},VIRTUAL+FIXUP
                ret

        method cancel
                ent p0 p1 p2:i0
                qcall dev/mouse/pcbus/cancel,(p0,p1,p2:i0),VIRTUAL+FIXUP
                ret

        defaultmethod
                entd
                parentclass
                ret

classend


tool 'dev/mouse/pcbus/_new',VP,0
        ;inputs
        ;       none
        ;outputs
        ;       p0 = instance pointer, else NULL if error
        ;
        ent -:p0

        cpy MH_SIZE+PCBUSMS_SIZE,i0
        qcall sys/kn/mem/allocdef,{i0:p0 i~}
         if.p p0!=NULL
                 ;make instance pointer
                add.p MH_SIZE,p0

                ;initialise the header for the object
                refclass p0,dev/mouse/pcbus/class
        endif
        ret

toolend


tool 'dev/mouse/pcbus/_init',VP,0
        ;inputs
        ;       p0 = instance pointer
        ;       p1 = ARGV
        ;       i0 = ARGC
        ;outputs
        ;       i0 = 0, else error code
        ;
        ent p0 p1 i0 : i0

        tracef  "dev/mouse/pcbus/init: starting %X\n", p0
        ;
        ; Initialise any instance variable defaults
        ;
        cpy     2, [p0+PCBUSMS_RESOLUTION]
        cpy     80, [p0+PCBUSMS_SAMPLERATE]
        clr     [p0+(PCBUSMS_ISR_VARS+PCBUSMSISR_NUM)]
        clr     [p0+(PCBUSMS_ISR_VARS+PCBUSMSISR_ERROR)]
        clr     [p0+(PCBUSMS_ISR_VARS+PCBUSMSISR_MSFLAGS)]
        clr     [p0+(PCBUSMS_ISR_VARS+PCBUSMSISR_XPOSN)]
        clr     [p0+(PCBUSMS_ISR_VARS+PCBUSMSISR_YPOSN)]
        clr     [p0+(PCBUSMS_ISR_VARS+PCBUSMSISR_BUTTONS)]
        cpy     pstate_init, [p0+(PCBUSMS_ISR_VARS+PCBUSMSISR_STATE)]
        clr     [p0+(PCBUSMS_ISR_VARS+PCBUSMSISR_LOCK_HANDLE_PORT_1)]
        clr     [p0+(PCBUSMS_ISR_VARS+PCBUSMSISR_LOCK_HANDLE_PORT_2)]
        clr     [p0+(PCBUSMS_ISR_VARS+PCBUSMSISR_LOCK_HANDLE_IRQ)]
        clr     [p0+(PCBUSMS_ISR_VARS+PCBUSMSISR_ISR_HANDLE)]

        ;
        ; Set default PC values for IRQ=12, port base=$60 & stride=4:
        ; Note that i/o buffer register is at port base &
        ; ctrl and status register is at port base + stride
        cpy     D_IRQ,[p0+(PCBUSMS_IRQ)]
        cpy     D_PORT_BASE,[p0+(PCBUSMS_ISR_VARS+PCBUSMSISR_PORT_BASE)]
        cpy     D_STRIDE,[p0+(PCBUSMS_ISR_VARS+PCBUSMSISR_STRIDE)]

        ;
        ; Setup input queues
        ;
        cpy.p   (p0+(PCBUSMS_ISR_VARS+PCBUSMSISR_CBUFF)), p2
        cpy.p   (p0+(PCBUSMS_ISR_VARS+PCBUSMSISR_INBUFFER)), p3
        circlebuff_init         p2, p3, PCBUSMS_INQSIZE, MS_SIZE

        ;
        ; Process any command line options
        ;
        cpy.p   options, p2
        qcall   lib/opt, {p1 p2 p0 i0:i0 i~}
        if i0=-1
                cpy     EINVAL, i0
                go      init_exit
        endif

        ;
        ; Initialise the mutex
        ;
        cpy.p   {p0+PCBUSMS_MTX}, p1
        cpy     MTX_PRIO|MTX_BPIP|MTX_SIGMASK, i0
        cpy     0, i1
        tracef  "dev/mouse/pcbus/init: Initialising mutex %X\n", p1
        qcall   sys/kn/mtx/init, {p1 i0 i1:i0}
        tracef  "dev/mouse/pcbus/init: Mtx/init returned %d\n", i0
        boolerrno i0, init_exit

        ;
        ; Book the io addresses to ensure shared access.  Do this
        ; before poking anything to the hardware.
        ;
        tracef  "dev/mouse/pcbus/init: lock io\n"
        ; i/o buffer address = PCBUSMSISR_PORT_BASE,
        cpy     [p0+PCBUSMS_ISR_VARS+PCBUSMSISR_PORT_BASE],i0       ;get port address to lock
        cpy     1, i1                                       ;number of bytes of io space to
lock
        cpy     LK_IO, i2                                   ;IO space
        qcall   dev/lockio, {i0 i1 i2:p1}
        bool    p1=-1, error_lkio
        cpy     p1, [p0+(PCBUSMS_ISR_VARS+PCBUSMSISR_LOCK_HANDLE_PORT_1)]

        ; control/status register address = PCBUSMSISR_PORT_BASE+PCBUSMSISR_STRIDE
         cpy
([p0+PCBUSMS_ISR_VARS+PCBUSMSISR_PORT_BASE]+[p0+PCBUSMS_ISR_VARS+PCBUSMSISR_STRIDE]),i0
                                                            ;get port address to lock
        qcall   dev/lockio, {i0 i1 i2:p1}
        bool    p1=-1, error_lkio
        cpy     p1, [p0+(PCBUSMS_ISR_VARS+PCBUSMSISR_LOCK_HANDLE_PORT_2)]

        ;
        ; Book the irq to ensure shared access.
        ;
        tracef  "dev/mouse/pcbus/init: lock irq\n"
        cpy     [p0+PCBUSMS_IRQ], i0          ;get irq to lock
        cpy     1, i1                                       ;number of irqs
        cpy     LK_IRQ, i2                                  ;IRQ space
        qcall   dev/lockio, {i0 i1 i2:p1}
        bool    p1=-1, error_lkirq
        cpy     p1, [p0+(PCBUSMS_ISR_VARS+PCBUSMSISR_LOCK_HANDLE_IRQ)]

        ;
        ; Load interupt service tool
        ;
        tracef "dev/mouse/pcbus/init: load isr\n"
        cpy.p   isrtoolname, p2
        cpy.p   (p0+PCBUSMS_ISR_VARS), p1
        cpy     PCBUSMS_ISR_VARS_SIZE, i0
        cpy     [p0+PCBUSMS_IRQ], i1          ;get irq to lock
        tracef  "dev/mouse/pcbus/_init: calling loadisr %x %x %d %d\n", p1, p2, i0, i1
        qcall   dev/loadisr, {p1 p2 i0 i1:p1 i~}
        tracef  "dev/mouse/pcbus/_init: loadisr returned %x\n", p1
        bool    p1=0, error_isr
        cpy     p1, [p0+(PCBUSMS_ISR_VARS+PCBUSMSISR_ISR_HANDLE)]

        ;
        ; Initialise AIO list
        ;
        cpy.p   (p0+(PCBUSMS_ISR_VARS+PCBUSMSISR_AIOHEAD)), p1
        initlist p1

        ;
        ; Initialise rest of instance variables and any hardware
        ;
        tracef  "dev/mouse/pcbus/init: initialise instance vars\n"
        cpy     [p0+PCBUSMS_SAMPLERATE], i0
        if i0<15
                cpy     10, i0
        elseif i0<30
                cpy     20, i0
        elseif i0<50
                cpy     40, i0
        elseif i0<70
                cpy     60, i0
        elseif i0<90
                cpy     80, i0
        elseif i0<150
                cpy     100, i0
        else
                cpy     200, i0
        endif
        cpy     i0, [p0+PCBUSMS_SAMPLERATE]

        ;
        ; Disable controller interrupts
        ;
        tracef  "dev/mouse/pcbus/init: disable controller interrupts\n"
        poll_aux_status        i0, i1, p1 ,p0
        aux_write_cmd  AUX_INTS_OFF, i1, i2, p1, p0

        ;
        ; Enable aux port
        ;
        tracef  "dev/mouse/pcbus/init: enable aux port\n"
        poll_aux_status i0, i1, p1, p0
        cpy     AUX_ENABLE, i0
        ; control/status register address = PCBUSMSISR_PORT_BASE+PCBUSMSISR_STRIDE
         cpy.p
([p0+PCBUSMS_ISR_VARS+PCBUSMSISR_PORT_BASE]+[p0+PCBUSMS_ISR_VARS+PCBUSMSISR_STRIDE]),p1
        outb_p  i0, p1

        ;
        ; Set sample rate and resultion
        ;
        aux_write_ack AUX_SET_SAMPLE, i1, i2, p1, p0
        aux_write_ack [p0+PCBUSMS_SAMPLERATE], i1, i2, p1, p0
        aux_write_ack AUX_SET_RES, i1, i2, p1, p0
        aux_write_ack [p0+PCBUSMS_RESOLUTION], i1, i2, p1, p0

        cpy     AUX_SET_SCALE11, i0
        if [p0+(PCBUSMS_ISR_VARS+PCBUSMSISR_MSFLAGS)]?BM_TWOTIMES
                cpy     AUX_SET_SCALE21, i0
        endif
        aux_write_ack i0, i1, i2, p1, p0
        aux_write_ack AUX_SET_STREAM, i1, i2, p1, p0
        poll_aux_status i0, i1, p1, p0

        ;
        ; Enable aux device
        ;
        tracef  "dev/mouse/pcbus/init: enable aux device\n"
        aux_write_dev AUX_ENABLE_DEV, i1, i2, p1, p0

        ;
        ; Enable controller interrupts
        ;
        aux_write_cmd AUX_INTS_ON, i1, i2, p1, p0
        poll_aux_status i0, i1, p1, p0

        cpy.l   CLOCKS_PER_SEC/10, l0
        tracef  "dev/mouse/pcbus/init: sleep for 0.1 sec\n"
        qcall   sys/kn/proc/sleep, {l0:i~}
        tracef  "dev/mouse/pcbus/init: sleep returned %d\n", i0

        ;
        ; Mark as in runing state
        ;
        cpy     PSTATE_0, [p0+(PCBUSMS_ISR_VARS+PCBUSMSISR_STATE)]

        clr i0         ; Exit OK

init_exit:
        tracef  "dev/mouse/pcbus/init: returning %d\n", i0
        ret
```

```
error_lkio:
        tracef  "Lock IO failed on port %d\n", i0
        cpy     ENOLCK, i0
        go      error_exit

error_lkirq:
        cpy     [p0+PCBUSMS_IRQ],i0
        tracef  "Lock IO failed on port %d\n", i0
        cpy     ENOLCK, i0
        go      error_exit

error_isr:
        cpy     [p0+PCBUSMS_IRQ],i0
        tracef  "Lock IO failed on port %d\n", i0
        cpy     EFAULT, i0
        go      error_exit

error_exit:
        if [p0+(PCBUSMS_ISR_VARS+PCBUSMSISR_ISR_HANDLE)] != 0
                cpy.p   [p0+(PCBUSMS_ISR_VARS+PCBUSMSISR_ISR_HANDLE)], p1          ;get irq
lock handle to unlock
                qcall   dev/unloadisr, {p1:-}
        endif
        cpy.p   [p0+(PCBUSMS_ISR_VARS+PCBUSMSISR_LOCK_HANDLE_IRQ)], p1                     ;get
irq lock handle to unlock
        if p1<>0
                qcall   dev/unlockio, {p1:-}
        endif
        cpy.p   [p0+(PCBUSMS_ISR_VARS+PCBUSMSISR_LOCK_HANDLE_PORT_2)], p1          ;get port
lock handle to unlock
        if p1<>0
                qcall   dev/unlockio, {p1:-}
        endif
        cpy.p   [p0+(PCBUSMS_ISR_VARS+PCBUSMSISR_LOCK_HANDLE_PORT_1)], p1          ;get port
lock handle to unlock
        if p1<>0
                qcall   dev/unlockio, {p1:-}
        endif

        cpy.p   {p0+PCBUSMS_MTX}, p1
        qcall   sys/kn/mtx/destroy, {p1:i~}
        ret

;       defcall a,dev/mouse/pcbusold/isr             ; Make sure the isr is available.
        __resource      dev/mouse/pcbus/isr           ; Make sure the isr is available.


aresolution:
        ent p0-p2 i0:p0-p1 i0

        qcall   lib/optval, {p0-p1:p0-p1}
        if p1!=0
                cpy.b   [p1], i1
                clr     i1
                if i1='$'
                        inc     p1
                        cpy     16, i1
                endif
                cpy.p   NULL, p3
                qcall   lib/strtol, {p1 p3 i1:l0}
                and     3, l0
                cpy     {l2i l0}, [p2+PCBUSMS_RESOLUTION]
        endif
        ret

asamplerate:
        ent p0-p2 i0:p0-p1 i0

        qcall   lib/optval, {p0-p1:p0-p1}
        if p1!=0
                cpy.b   [p1], i1
                clr     i1
                if i1='$'
                        inc     p1
                        cpy     16, i1
                cpy.p   NULL, p3
                qcall   lib/strtol, {p1 p3 i1:l0}
                cpy     {l2i l0}, [p2+PCBUSMS_SAMPLERATE]
        endif
        ret

twotimes:
        ent p0-p2 i0:p0-p1 i0

        or      M_TWOTIMES, [p2+(PCBUSMS_ISR_VARS+PCBUSMSISR_MSFLAGS)]
        add     4, p0
        ret

exponent:
        ent p0-p2 i0:p0-p1 i0

        or      M_EXP, [p2+(PCBUSMS_ISR_VARS+PCBUSMSISR_MSFLAGS)]
        add     4, p0
        ret

lefthand:
        ent p0-p2 i0:p0-p1 i0

        or      M_LEFTHAND, [p2+(PCBUSMS_ISR_VARS+PCBUSMSISR_MSFLAGS)]
        add     4, p0
        ret

intelli:
        ent p0-p2 i0:p0-p1 i0

        or      M_INTELLI, [p2+(PCBUSMS_ISR_VARS+PCBUSMSISR_MSFLAGS)]
        add     4, p0
        ret

; store value of port base
port_base:
        ent     p0-p2 i0 : p0-p1 i0

        qcall   lib/optval,(p0 p1 : p0 p1)                  ;extract parameter
        if.p    p1 ne NULL
                cpy.b [p1],i1
                if i1='$'
                        inc p1
                        cpy 16,i1                           ;use base 16
                else
                        cpy 0,i1                            ;context sensitive mode
                endif
                cpy.p NULL,p3                               ;don't store result
                qcall lib/strtol,(p1 p3 i1: l0)
                cpy     l2i l0,[p2 + (PCBUSMS_ISR_VARS+PCBUSMSISR_PORT_BASE)]
        endif
        ret

; store irq
interrupt:
        ent     p0-p2 i0 : p0-p1 i0

        qcall   lib/optval,(p0 p1 : p0 p1)                  ;extract parameter
        if.p    p1 ne NULL
                cpy.b [p1],i1
                if i1='$'
                        inc p1
                        cpy 16,i1                           ;use base 16
                else
                        cpy 0,i1                            ;context sensitive mode
                endif
                cpy.p   NULL,p3                             ;don't store result
                qcall   lib/strtol,(p1 p3 i1 : l0)
                cpy     l2i l0,[p2 + PCBUSMS_IRQ]
        endif
        ret


; store register stride
stride:
        ent     p0-p2 i0 : p0-p1 i0

        qcall   lib/optval,(p0 p1 : p0 p1)                    ;extract parameter
        if.p    p1 ne NULL
                cpy.b [p1],i1
                if i1='$'
                        inc p1
                        cpy 16,i1                             ;use base 16
                else
                        cpy 0,i1                              ;context sensitive mode
                endif
                cpy.p   NULL,p3                               ;don't store result
                qcall   lib/strtol,(p1 p3 i1: l0)
                cpy     l2i l0,[p2 + (PCBUSMS_ISR_VARS+PCBUSMSISR_STRIDE)]
        endif
        ret



        data
options:
        ;options table for command line processing
        dc.i 1,1
        dc.i ('l'),lefthand
        dc.i ('r'),aresolution
        dc.i ('2'),twotimes
        dc.i ('s'),asamplerate
        dc.i ('e'),exponent
        dc.i ('w'),intelli

        ; command line irq, port base and register stride
        dc.i ('i'),interrupt
        dc.i ('p'),port_base
        dc.i ('t'),stride
        dc.i 0


isrtoolname:
        dc.b 'dev/mouse/pcbus/isr', 0

toolend


tool 'dev/mouse/pcbus/_deinit',VP,0
        ;inputs
        ;       p0 = instance pointer
        ;outputs
        ;       i0 = deinit code
        ;
        ent p0 : i0

        tracef  "dev/mouse/pcbus/_deinit: started %x\n", p0
        ;lock instance
        cpy {p0+PCBUSMS_MTX},p1
        qcall sys/kn/mtx/lock,{p1:i0}
        boolerrno i0,deinit_exit

        tracef  "dev/mouse/pcbus/_deinit: wait for interrupt to get to start state\n"
        ;wait for interupt to get to start state
        cpy {[p0+(PCBUSMS_ISR_VARS+PCBUSMSISR_MSFLAGS)] bset
BM_QUIT},[p0+(PCBUSMS_ISR_VARS+PCBUSMSISR_MSFLAGS)]
        if [p0+(PCBUSMS_ISR_VARS+PCBUSMSISR_STATE)]!=PSTATE_0
                repeat
                        tracef  "dev/mouse/pcbus/_deinit: calling sys/kn/proc/deschedule\n"
                        qcall sys/kn/proc/deschedule,{-:-}
                until [p0+(PCBUSMS_ISR_VARS+PCBUSMSISR_STATE)]=PSTATE_0
        endif

        ;disable interupts
        tracef  "dev/mouse/pcbus/_deinit: disable interrupts\n"
        aux_write_dev AUX_DISABLE_DEV, i0, i1, p1, p0
        aux_write_cmd AUX_INTS_OFF, i0, i1, p1, p0
        poll_aux_status i0, i1, p1, p0
        cpy AUX_DISABLE, i0
         cpy.p
([p0+PCBUSMS_ISR_VARS+PCBUSMSISR_PORT_BASE]+[p0+PCBUSMS_ISR_VARS+PCBUSMSISR_STRIDE]), p1
```

```
        poll_aux_status i0, i1, p1, p0

        tracef  "dev/mouse/pcbus/_deinit: unload isr\n"
        ;
        ; Unload isr tool
        ;
        cpy     [p0+(PCBUSMS_ISR_VARS+PCBUSMSISR_ISR_HANDLE)], p1
        tracef  "dev/mouse/pcbus/_deinit: calling unloadisr %x\n", p1
        qcall   dev/unloadisr,{p1:-}

        tracef  "dev/mouse/pcbus/_deinit: unlock locks\n"
        ;unlock all locks
        cpy.p [p0+(PCBUSMS_ISR_VARS+PCBUSMSISR_LOCK_HANDLE_IRQ)],p1        ;get irq lock
handle to unlock
        tracef  "dev/mouse/pcbus/_deinit: calling unlockio %x\n", p1
        qcall dev/unlockio,{p1:-}

        cpy.p [p0+(PCBUSMS_ISR_VARS+PCBUSMSISR_LOCK_HANDLE_PORT_2)],p1     ;get port lock
handle to unlock
        tracef  "dev/mouse/pcbus/_deinit: calling unlockio %x\n", p1
        qcall dev/unlockio,{p1:-}

        cpy.p [p0+(PCBUSMS_ISR_VARS+PCBUSMSISR_LOCK_HANDLE_PORT_1)],p1     ;get port lock
handle to unlock
        tracef  "dev/mouse/pcbus/_deinit: calling unlockio %x\n", p1
        qcall dev/unlockio,{p1:-}

        ;mark as deinited
        cpy 1,i0

        tracef  "dev/mouse/pcbus/_deinit: unlock instance\n"
        ;unlock instance
        cpy {p0+PCBUSMS_MTX},p1
        qcall sys/kn/mtx/unlock,{p1:i~}
        qcall sys/kn/mtx/destroy,{p1:i~}

deinit_exit:
        ret

toolend

tool 'dev/mouse/pcbus/close', VP, 0
        ;inputs
        ;       p0 = instance pointer
        ;       p1 = handle (ignored)
        ;
        ;outptus
        ;       i0 = 0 = success
        ;
        ;Calls the cancel method for all AIOs in PCBUSMSISR_AIOHEAD queue.
        ;The cancel method removes the AIO from the queue and deals with
        ;mutex and interrupt protection.  The cancel method can return an error
        ;
        ent     p0 p1:i0

        tracef  "dev/mouse/pcbus/close: open count = 0 so cancelling any AIOs\n"

        clr.i   i0                                          ;Indicate device closed
successfully

        cpy.p   p0+(PCBUSMS_ISR_VARS+PCBUSMSISR_AIOHEAD), p2 ;Reference AIO queue
        while   [[p2 + LH_HEAD] + LN_SUCC] != NULL          ;While there's an AIO at the head
of the queue
        ;
                cpy.p   [p2],p3
                qcall   dev/mouse/pcbus/cancel,(p0 p1 p3 : i~)      ;Call cancel method on
AIO.
                                                            ;No important error codes are
returned
        ;                                                   ;unless some horrible programming
error occures
        endwhile
        ret
toolend
```

```
tool 'dev/mouse/pcbus/reada', VP, 0
        ;inputs
        ;       p0 = instance pointer
        ;       p1 = handle
        ;       p2 = data buffer pointer
        ;       p3 = AIO pointer
        ;       i0 = number of bytes to read
        ;outputs
        ;       i0 = bytes read, else error code
        ent p0-p3 i0:i0

        tracef  "dev/mouse/pcbus/reada: started with %x %x %x %x %d\n", p0, p1, p2, p3, i0

        if i0 == 0
                qcall dev/iocomplete, (p3, 0.i : i~)
                clr.i i0
                go      reada_exit
        endif

        if i0<MS_SIZE
                tracef "dev/mouse/pcbus/reada: buffer too small, needs to be at least %x
bytes\n", MS_SIZE
                cpy     EINVAL, i0
                go      reada_exit
        endif


        ;
        ; Lock instance
        ;
        cpy     (p0+PCBUSMS_MTX), p4
        tracef  "dev/mouse/pcbus/reada: set lock %x\n", p4
        qcall   sys/kn/mtx/lock, {p4:i0}
        boolerrno i0, reada_exit
        tracef  "dev/mouse/pcbus/reada: got mtx\n"

        ;
        ; See if there is any data in the buffer
        ;
        clr     i0
        qcall   sys/pii/int_off, {-:i4}
        cpy.p   (p0+(PCBUSMS_ISR_VARS+PCBUSMSISR_CBUFF)), p5
        circlebuff_front p5, i1
        circlebuff_back p5, i2
        if i1 != i2
                tracef  "dev/mouse/pcbus/reada: Data available\n"
                ;
                ; Get the data and update circular buffer indexes
                ;
                circlebuff_readpos p5, i2, p4

                circlebuff_elementsize p5, i3
                cpbb    p4, p2, i3

                tracef  "dev/mouse/pcbus/reada: updating buffers\n"
                circlebuff_frontnext p5, i2

                cpy     i3, i0

                if [p0+(PCBUSMS_ISR_VARS+PCBUSMSISR_ISRHELD)] != 0
                        circlebuff_front p5, i1
                        circlebuff_back p5, i2
                        if i1 = i2
                                ;
                                ; Queue empty - restart ISR
                                ;
                                tracef  "dev/mouse/pcbus/reada: starting isr\n"
                                clr     [p0+(PCBUSMS_ISR_VARS+PCBUSMSISR_ISRHELD)]
                                cpy     (p0+PCBUSMS_ISR_VARS), p4
                                qcall   dev/mouse/pcbus/isr, {p4:-}
                        endif
                endif
        endif

        if i0 != 0
```

```
                qcall   sys/pii/int_restore, {i4:-}

                tracef  "dev/mouse/pcbus/reada: calling iocomplete with %x %d\n", p3, i0
                qcall   dev/iocomplete, {p3 i0:i0}
        else
                if p3 = NULL
                        cpy     EAGAIN, i0
                else
                        cpy.p   p2, [p3+AIO_DEVPTR1]
                        reftool dev/mouse/pcbus/cancel,p4
                        cpy.p   p4,[p3+AIO_CANCEL]
                        tracef  "dev/mouse/pcbus/reada: adding AIO to list\n"
                        cpy.p   (p0+(PCBUSMS_ISR_VARS+PCBUSMSISR_AIOHEAD)), p4
                        addtail p4, p3, p5
                endif

                qcall   sys/pii/int_restore, {i4:-}
        endif

        ;
        ; Unlock instance
        ;
        tracef  "dev/mouse/pcbus/reada: unlocking mutex\n"
        cpy.p   (p0+PCBUSMS_MTX), p4
        qcall   sys/kn/mtx/unlock, {p4:i~}

reada_exit:
        ret

toolend

tool 'dev/mouse/pcbus/info',VP,0
        ;inputs
        ;       p0 = instance pointer
        ;       p1 = handle
        ;       p2 = data buffer pointer
        ;       i0 = size of buffer
        ;outputs
        ;       i0 = bytes returned
        ;
        ent p0 p1 p2 i0:i0

        ;smaller of buffer size or info block
        cpy (devinfoend-devinfo),i1
        if i0>i1
                cpy i1,i0
        endif

        ;copy info block to buffer
        cpy.p (devinfo),p3
        cpbb p3,p2,i0

        ;
        ; Put in actual current values for the io range and irq
        ; overwrite the fields just copied to the callers memory
        ; (range check, caller may request 1st 4 bytes to obtain
        ; length).
        ; i/o buffer address = PCBUSMSISR_PORT_BASE,
        ; control/status register address = PCBUSMSISR_PORT_BASE+PCBUSMSISR_STRIDE
        ;
        cpy [p0+(PCBUSMS_ISR_VARS+PCBUSMSISR_MSFLAGS)],i2
        if i2?M_INTELLI
                or DFI_WHEEL,[p2+8]           ;Pointer has a wheel
        endif

        ; Insert value of irq and address of control & buffer registers
        bcn     i0 lt (io_off+4),info_exit
        cpy     [p0 + (PCBUSMS_ISR_VARS+PCBUSMSISR_PORT_BASE)],[p2 + io_off]       ;patch
buffer reg
        bcn     i0 lt (io_off+12),info_exit
        cpy     ([p0 + (PCBUSMS_ISR_VARS+PCBUSMSISR_PORT_BASE)] +
[p0+(PCBUSMS_ISR_VARS+PCBUSMSISR_STRIDE)]),[p2 + (io_off+8)]    ;patch ctrl reg
        bcn     i0 lt (irq_off+4),info_exit
        cpy     [p0+PCBUSMS_IRQ],[p2 + irq_off]
        ;patch irq
        ret

data
devinfo:
        ;size of info data
        dc.i devinfoend-devinfo

        ;family device type
        dc.i DF_POINTER
        dc.i 0

        ;use default values for irq & address of control & buffer registers
irq_off=16
        dc.i DD_IRQ,D_IRQ
io_off=24
        ; i/o buffer address = D_PORT_BASE,
        ; control/status register address = D_PORT_BASE+D_STRIDE
        dc.i DD_IO,D_PORT_BASE
        dc.i DD_IO,D_PORT_BASE+D_STRIDE
        dc.i DD_TERMINATOR
         revision "PC Bus Mouse Driver Version ", $Revision: 1.21 $, 0

devinfoend:

toolend


tool 'dev/mouse/pcbus/cancel', VP, 0
        ;inputs
        ;       p0 = driver instance pointer
        ;       p1 = handle (ignored)
        ;       p2 = AIO pointer
        ;outputs
        ;       i0 = zero or error code
        ;            note that no data is ever returned as half a mouse packet is useless
        ;
        ent p0 p1 p2 : i0

        tracef  "dev/mouse/pcbus/cancel: requesting mutex lock\n"
        cpy.p   (p0 + PCBUSMS_MTX),p1
        qcall   sys/kn/mtx/lock, (p1:i0)
        boolerrno i0, mutex_error
        tracef  "dev/mouse/pcbus/cancel: granted mutex lock\n"

        qcall   sys/pii/int_off,(-:i1)               ;Protect this to prevent both cancel
method
                                                     ;  and the ISR calling dev/iocomplete

        reftool dev/mouse/pcbus/cancel,p1            ;Is this method associated with this AIO?
        if.p    [p2 + AIO_CANCEL] == p1
        ;                                            ;YES
                clr.i   [p2 + AIO_CANCEL]            ;Prevent another cancel taking place

                cpy.p   p2,p1                        ;Remove AIO from PCBUSMSISR_AIOHEAD list
                remove  p1,p3                        ;  (destroys p1 & p3, hence copying)

                qcall   sys/pii/int_restore,(i1:-)   ;ISR can't touch AIO when not in queue
                                                     ;  so turn ints back on
                cpy.i   ECANCELED,i0
                qcall   dev/iocomplete,(p2,i0:i~)    ;Complete AIO with ECANCELLED error code
                clr.i   i0                           ;Indicate success of cancel routine

                tracef  "dev/mouse/pcbus/cancel: cancelled the AIO\n"
        ;
        else
        ;                                            ;NO
                qcall   sys/pii/int_restore,(i1:-)   ;Ints back on

                cpy.i   EINVAL,i0                    ;Indicate cancel didn't happen

                tracef  "dev/mouse/pcbus/cancel: AIO not cancelled\n"
        ;
        endif

        cpy.p   (p0 + PCBUSMS_MTX),p1
        gcall   sys/kn/mtx/unlock, (p1:i1)
        boolerrno i1,mutex_error

        tracef  "dev/mouse/pcbus/cancel: unlocked mutex\n"
        go      exit

mutex_error:
        tracef  "dev/mouse/pcbus/cancel: cannot unlock mutex\n"
        cpy.i   EINVAL,i0
        go      exit

exit:
        ret

toolend


tool 'dev/mouse/pcbus/isr', VP, 0
        ;inputs
        ;       p0 = interrupt structure
        ;outputs
        ;       none
        ent p0:-

isrstart:
        cpy.p   (p0+PCBUSMSISR_CBUFF), p6
        if [p0+PCBUSMSISR_STATE] = PSTATE_0
                ;
                ; Check for room in the buffer before we try to read it
                ;
                cpy.p   [p0+(PCBUSMSISR_AIOHEAD+LH_HEAD)], p1
                if [p1+LN_SUCC] = NULL
                        circlebuff_noroom p6, i1
                        cpy     i1, [p0+PCBUSMSISR_ISRHELD]
                endif
        endif

getevent:
        ;
        ; We have room so get an event from the mouse
        ;
        if [p0+PCBUSMSISR_STATE] <> PSTATE_0
                bool [p0+PCBUSMSISR_MSFLAGS]?BM_QUIT, finished
        endif

        gos     get_byte, {p0:i0 i1}
        bool    i1 = 0, exit

        cpy     [p0+PCBUSMSISR_STATE], i1
        bool    i1 = PSTATE_INIT, state_init

        bool    i1 = PSTATE_0, state0
        bool    i1 = PSTATE_1, state1
        bool    i1 = PSTATE_2, state2

        inc     [p0+PCBUSMSISR_ERROR]
        cpy     PSTATE_0, [p0+PCBUSMSISR_STATE]
        go      exit

state_init:
        cpy     [p0+PCBUSMSISR_PS2DATA], i0
        go      exit

state0:
        cpy     i0, [p0+PCBUSMSISR_INTDATA0]
        cpy     PSTATE_1, [p0+PCBUSMSISR_STATE]
        go      exit

state1:
        cpy     i0, [p0+PCBUSMSISR_INTDATA1]
        cpy     PSTATE_2, [p0+PCBUSMSISR_STATE]
        go      exit

state2:
        cpy     i0, [p0+PCBUSMSISR_INTDATA2]
```

```
process:
        ;
        ; Reset to state 0
        ;
        cpy     PSTATE_0, [p0+PCBUSMSISR_STATE]

        ;
        ; Process the data
        ;
        cpy     [p0+PCBUSMSISR_INTDATA0], i0
        and     3, i0
        cpy     i0, [p0+PCBUSMSISR_BUTTONS]
        bool    [p0+PCBUSMSISR_MSFLAGS]?BM_EXP, exp1

        cpy     (b2i [p0+PCBUSMSISR_INTDATA1]), i0
        cpy     i0, [p0+PCBUSMSISR_XPOSN]
        cpy     (b2i [p0+PCBUSMSISR_INTDATA2]), i0
        neg     i0
        cpy     i0, [p0+PCBUSMSISR_YPOSN]
        go      finished

exp1:
        cpy     (b2i [p0+PCBUSMSISR_INTDATA1]), i0
        if i0<0
                cpy     (i0*i0), i0
                neg     i0
        else
                cpy     (i0*i0), i0
        endif
        add     i0, [p0+PCBUSMSISR_XPOSN]

        cpy     (b2i [p0+PCBUSMSISR_INTDATA2]), i0
        if i0<0
                cpy     (i0*i0), i0
                neg     i0
        else
                cpy     (i0*i0), i0
        endif
        sub     i0, [p0+PCBUSMSISR_YPOSN]

finished:
        ;
        ; Check for waiting AIOs
        ;
        cpy.p   [p0+(PCBUSMSISR_AIOHEAD+LH_HEAD)], p1

        if [p1+LN_SUCC] = NULL
                ;
                ; No AIOs so put into circular buffer
                ;
                circlebuff_back p6, i1
                circlebuff_start p6, p3

                if [p0+PCBUSMSISR_ISRHELD] = 0
                        cpy.p   (p3+i1), p3
                endif
                clr.p   p1
        else
                ;
                ; Put data into AIO buffer and remove AIO from list
                ;
                cpy.p   p1, p4
                cpy.p   [p1+AIO_DEVPTR1], p3
                remove  p4, p5
        endif

        ;
        ; Copy the data
        ;
        ; Scale deltas to the same order of magnitude as plane coordinates.
        cpy     ((ld.i [p0+PCBUSMSISR_XPOSN]) lsl 7), [p3+MS_X]
        cpy     ((ld.i [p0+PCBUSMSISR_YPOSN]) lsl 7), [p3+MS_Y]
        cpy     [p0+PCBUSMSISR_BUTTONS], [p3+MS_BUTTONS]
        qcall   sys/kn/time/get, (-:l0)
        cpy.l   l0, [p3+MS_TIMESTAMP]
        if p1 = NULL
                circlebuff_backnext p6, i1
                circlebuff_rollup p6, i1
        else
                clr.p   [p1+AIO_CANCEL]
                circlebuff_elementsize p6, i0
                qcall   dev/iocomplete, {p1 i0:i~}
        endif

        go      isrstart

exit:
        ret

;
;-----------------------------------------------------------------------------------
;
; get a byte of data from the bus port
;
get_byte:
        ;inputs
        ;       p0 = instance pointer
        ;outputs
        ;       i0 = byte read from mouse (if i1=1)
        ;       i1 = 0 if no data read from mouse, 1 if data read from mouse
        ;
        ent p0 :i0 i1

        clr     i1

        ;
        ; Read status register
        ;
        ; control/status register address = PCBUSMSISR_PORT_BASE+PCBUSMSISR_STRIDE
         cpy.p   ([p0+PCBUSMSISR_PORT_BASE]+[p0+PCBUSMSISR_STRIDE]),p1
        ioin.b  p1, i0
        bool i0!?0, nodata
        bool i0!?5, nodata

        ;
        ; Read data
        ;
        ; i/o buffer address = PCBUSMSISR_PORT_BASE,
        cpy.p   [p0+PCBUSMSISR_PORT_BASE],p1
        ioin.b  p1, i0
        inc     i1

nodata:
        ret

toolend

.end
```