Initial Experience Porting Linux Device Drivers to Cyclone

Nikhil Swamy Michael Hicks

June 29, 2007

1 Placeholder

1.1 8139too.cyc, a RealTek RTL-8139 Fast Ethernet driver for Linux

1.1.1 Overview of driver behavior

- **Module initialization**

  The module is initialized using the standard scheme for pci devices; i.e, initialization using module_init / module_exit, with pci_module_init called on a driver table that is statically partially populated. Similarly, module exit just unregisters the pci driver.

  Pci probing is done by rtl8139_init_one. The function is registered as a callback in the pci driver tables. The main components of device probing are classified as follows:

  1. Allocating the Ethernet Device

     The ether device structure is colocated with the general net device structure. The netdevice interface provides an "alloc_etherdev" function that allocates space for both the ethernet device as well as the net_device. In the case of linux 2.6, the rules for allocating/deallocating a netdevice are pretty stringent. (See the notes in Documentation/networking/netdevices.txt for allocation rules for netdevs). This is primarily because of the introduction of sysfs

(/sys/class/net etc.) in 2.6 which can allow the module to be removed while still holding an entry for that device in sysfs.

2. Initializing IO memory
   The pci interface provides methods (`resource_start` and `resource_end`) that return the range of bus addresses assigned to this device. These addresses correspond to memory resident on the device, and maybe thought of as the control registers of the device. This memory is ioremapped (dma style) before being used by the driver. The ioremapped address is what is stored in the drivers private data structure.
3. Initializing Card State
   The driver writes various values to the card registers, and reads the resulting card state to discover information such as chip version, clock speed etc. Once this has been done the state of the card is initialized to some start state.
4. Populating Structures and Registration
   Setting callbacks and fields on the netdevice structure, and in the mii interface of the etherdevice (media independent interface) takes place. The structures are registered with their respective controllers i.e `register_net_dev` etc.
5. Failure Handling
   In case of failure of any of these steps, a cleanup function releases pci alocated io memory, unmaps any dma, and frees the allocated netdevices etc.

- **Post Initialization – Main Functional Path**

  After initialization the netdevice controller takes over the driver. The functions registered in the netdevice structure serve as callbacks for the netdevice structure. The functions are: `open, start_xmit, close, stats, rx_mode, ioctl, tx_timeout`. Each of these are described here. See networking/netdevices.txt in the kernel documentation for synchronization rules for each of these functions.

  1. `int rtl8139_open (net_device_t dev)`
     This routine is called by netdevice controller in the case of something like "ifup eth0". That makes this routine the main entry point of the driver. Its functions are summarized below.

(a) Grab an irq line, and register an interrupt handler. In this case the interrupt handler is `rtl8139_interrupt`. As with all pci enabled devices, this is a shared irq line.

(b) Dma allocate io memory using pci_alloc_consistent. This memory is used for the transmit and receipt buffers – stashed in the drivers private data, along with the control register addresses. It is memory shared between the device and processor, and both the raw bus address as well as the kernel virtual address is returned by the pci method.

(c) Calls hw_start. This function writes repeatedly to various control registers. Also does some mii stuff to set modes etc. This function finally calls netif_start_queue, which allows the rest of the netdevice scheduler to proceed.

(d) Initializes a kernel thread which is an infinite loop that wakes itself up periodically. It is used essentially to some fine tuning of clocks, echo due to impedance of cables etc.

(e) In case of failure, tx/rx bufs are released. Irq line is returned.

2. `int netdev_ioctl(net_device_t dev, struct ifreq *rq, int cmd)`

   Once the card is initialized and scheduled properly, it can accept requests from user space. This function allows for the testing and setting of various parameters condition on the cmd and rq parameters. This is also an entry point to the underlying mii interface.

   The parameter `rq` is a pointer to user memory. The size of the memory pointed to (the actual structure) is determined by the first word of that memory. This tag word is the command that is issued to the device. Depending on the value of the command, values are either written to user space, or read from user space into the device's control registers, or both.

3. `int rtl8139_start_xmit (struct sk_buff *skb, net_device_t dev)`

   Copies the socket buffer to the tx_buf iomemory. Uses memset to zero out the buffer first in some cases. More importantly, frees the sk_buff by calling dev_kfree_skb. In the case where the packet is too big, it is freed anyway i.e the packet is dropped. The card is notified of the packet awaiting transmission by writing to a control register. In case of some failure by the card, the netdevice

is removed from the queue ... i.e it is effectively suspended. This will result eventually in tx_timeout being called.

4. `void rtl8139_interrupt (int irq, net_device_t dev, struct pt_regs *regs)`

   This is the main interrupt handler, for all interrupts raised on this driver's irq line. It reads the interrupt status from a control register. The status signals one of the following: transmit completion, or transmit error; a packet receipt; some other error condition.

   (a) In the case of a transmit interrupt the handler must clean up after the tx thread. This cleanup simply logs errors if they have occured, and updates the count of still outstanding tx transactions.

   (b) Error checking in the case of a receipt interrupt involves ensuring that the size of the packet received is within an acceptable range. Once this is confirmed, a socket buffer (sk_buff) is allocated using dev_alloc_skb. Data in the consistently allocated rx_bufs is copied to the sk_buf using `eth_copy_and_sum` which is a variant of memset. The skb is then handed off to the net-device interface using netif_rx. Freeing the skb becomes the responsibility of netif. This copy/skb-handoff process is repeated until the card's rx buffer is empty.

   (c) In the exceptional case, driver error counts are updated. In the case of pci errors, the card status is updated.

5. `void rtl8139_tx_timeout (net_device_t dev)`

   Cancels all outstanding transactions. Disables interrupts. Restarts hardware.

6. `struct net_device_stats *rtl8139_get_stats (net_device_t dev)`

   Returns accumulated statistics (error counts etc.)

7. `int rtl8139_close (net_device_t dev)`

   Called in the case of "ifdown eth0" or something equivalent. It removes the device from the netif scheduler. Kills the fine tuning thread that was started by open. Clears interrupt masks for the card. Relinquishes the interrupt line. Frees IO-memory in the tx/rx buffers. Importantly, it does not release the iomemory for the card's control registers – i.e the ioremap'd bus start/end obtained from the pci interface is not unmapped yet. That is, close

function does not mean that the module is to be removed. That final step is performed by the following function.

8. `void __devexit rtl8139_remove_one (pci_dev_t pdev)`
   This function is registered with the pci interface, and is called when the module is unloaded. It iounmaps the control reg. memory, unregisters the netdev with the netif. And finally, it frees the network devices allocated in the pci probe function.

1.1.2 Cyclone safety mechanisms

Several features of Cyclone were used to obtain a type-safe port of this driver. [1]

1. Fat pointers for IO Memory

   Much of the code in this driver consists of writing to iomemory. This memory is acquired from the pci interface and then ioremapped to kernel virtual addresses. The orginal C driver used the macros readb and writeb to access these buffers. The signatures of these macros are effectively

   ```
   u8 readb(unsigned long ioaddr, unsigned offset);
   void writeb(unsigned long ioaddr, unsigned offset, u8 val);
   ```

   To guarantee the safety of the pointer arithmetic in these buffers, pointers to iomemory were implemented as fat pointers. Functions such as pci_resource_start / pci_resource_end which allocate io-memory were wrapped (see below) to return fat pointers instead.

   The macros readb/writeb were replaced by inlined extern C functions such as the following.

   ```
   inline void CYC_WRITEB(u8 ?@nozeroterm fat_ioaddr, int offset, u8 val8){
     void *ioaddr = fat_ioaddr.base;
     if(ioaddr + reg >= (void*)fat_ioaddr.last_plus_one)
       _throw_arraybounds();
     writeb ((val8), ioaddr + (reg));
   }
   ```

[1] This was the first driver we ported. As such, some of the techniques used were refined for subsequent drivers. One glaring omission is the handling of the allocation and freeing of the `net_device` structure. A scheme to handle this better is outlined in Section ??. But, at the time of writing, this has not yet been implemented.

Note that these functions are declared as extern "C" and thus are not typechecked by Cyclone. This is required since the function body still uses `writeb`, which cannot be proved safe by Cyclone. The intention is for such functions to be part of a trusted library of io utilities that are provided with Cyclone for the development of kernel modules.

2. Dependent Types for tx/rx Buffers

   Fat pointers were used primarily for io-memory that is resident on the card. Processor/device shared buffers that are allocated by pci_alloc_consistent (or some other consistent DMA scheme), were instead represented using dependent types. Such memory is not accessed to using the readb/writeb scheme.

   For instance, in the case of the transmission buffers, a large chunk of memory is allocated. This is then split into a number of smaller chunks of equal size, one for each of several transmission buffers that the driver allows to be filled concurrently. Accesses to these buffers are done using functions such as memset, where it is sufficient to prove that the size of the memory being copied is no bigger than the space of the destination. In addition, by encoding these pointers as dependent types several array bounds checks can be amortized into perhaps even a single check.

   The tx_bufs and rx_bufs have the following type:

```
typedef struct bounded_iobuf {
  <`i::I>
  u8 *valueof(`i)@nozeroterm buf;
  tag_t<`i> len;
} bnd_buf_t;
```

3. Wrappers for pci resource allocation

   Pci resource allocation functions were wrapped by Cyclone functions so that fat pointers, or dependent types can be safely manufactured from them. For instance, the pci_alloc_consistent has the following wrapper:

```
bnd_buf_t pci_alloc(pci_dev_t hwdev, size_t tag, dma_addr_t *dma_handle) {
  u8 *virt_addr = pci_alloc_consistent(hwdev, tag, dma_handle);
  bnd_buf_t res = bounded_iobuf{.buf = virt_addr, .len = tag};
  return res;
}
```

From a bnd_buf_t it is easy to manufacture a fat pointer using the standard Core::mkfat Cyclone library function. Alternatively, the dependent type can be used directly to prove the safety of io pointer dereferences.

4. Overriding Type Declarations

   Many kernel data structures (such as net_device) allow for a driver to store a pointer to a private data structure within it. This is a classic example of the use of the "void *" type in C to encode parametric polymorphism. To eliminate the Cyclone unsupported downcast from void* to the expected private data type, a number of kernel data structures had to explicitly be declared polymorphic. For instance, the net_device structure was declared as follows:

```
struct net_device <`a::A> {
        ...
        `a *priv;
        ...
        struct net_device<`a> *next;
        ...
        int (*open)(struct net_device<`a> *dev);
        ...
};
```

   Here, the "priv" field is a pointer to the driver's private data. The declaration of the function pointer field "open" is also suitably updated so that in the driver function "rtl8139_open" we can safely eliminate the downcast of "priv" from "void*".

   Note however the type of the field "next" is inaccurate, since the next network device in the chain of devices is, most likely, controlled by some other driver which will instantiate the "priv" field differently. However, this next field is never accessed by any driver. Thus we can maintain the illusion of a homogenous list safely.

   The number of types that had to be overridden in this manner was quite large. These included the pci_dev and its related structuers, the structures related to the mii interface.

   Other types were overridden too. Examples of these include adding requires clauses to functions such as memset to ensure that the buffers

being copied were of appropriate size. Non-null checks were imposed by overriding function declaration to contain "@" pointers instead of "*" pointers etc.

5. Regions for Short-lived Allocations

   There were some cases in the ioctl routines that heap allocated short-lived data structures. These were instead replaced with lexical regions. An alternative approach was to implement these using unique pointers, since this does not entail the additional cost of allocating an entire region page. Both approaches were implemented.

6. Tagged Union for ioctl User Memory

   It should be clear from the description of the "netdev_ioctl" function in the previous section, that the natural representation of the user request is as a tagged union. The C declaration of the type, however, does not even represent the request as a normal union. The driver simply examines the first word of the user space memory and conditional on that word casts the pointer to some appropriate type. In this case, the representation of the user-space allocated memory is completely determined by an implicit definition of the protocol.

   To support a type-safe implementation of this function, we had to provide a C function that would produce a Cyclone tagged union after examining user-space memory. Once this tagged union was generated, the rest of the ioctl could use the union in a type safe manner.

7. Wrapping Functions with cyc_call Exception Handlers

   Cyclone functions can throw exceptions. If a Cyclone function is called from C without first installing an top-level exception handler, then the stack unwinding process fails. Obviously, in case an uncaught exception is thrown, the result is a kernel oops.

   To avoid this, we must ensure that every Cyclone function in the driver that is exposed directly to the kernel must never throw an exception. In this driver, this was achieved by wrapping each Cyclone function by a C function that explicitly installs a default exception handler. For instance, the "rtl8139_open" is exposed to the kernel since it is registered as a callback with the netdevice interface.

1.1.3 Cyclone modifications/enhancements

1. Overriding Type Declarations As described in the previous section, a considerable proportion of the changes made to the C driver involved assigning a meaningful Cyclone type to C type declarations. For this purpose, the Cyclone language was equipped with the construct illustrated in Figure **??**.

   The block of code within the `extern ‘‘C include’’` contains the C declarations that are to be imported for use in the Cyclone program. The `cyclone_override` block allows the user to modify the C type declaration. The Cyclone compiler ensures that the overridden type is *representation consistent* with the original C declaration.

   In this case, a polymorphic type variable has been introduced in the declaration of `net_device`. The introduction of this variable has an impact on the declaration of other C types too. For instance, the function `netif_start_queue` must be updated to reflect the new type variable. Similarly, the structure `mii_info` must also be updated to reflect the type variable that now appears in its `dev` field. Cyclone now supports two methods of reflecting these new type variables in the hierarchy of type declarations.

   In the first mode (the default) Cyclone updates the declarations using a form of “most-general” type variable inference. In this case, the updates would be as follows:

   ```
   struct mii_info<‘a::A> {
     int phy_id;
     struct net_device<‘a> *dev;
     struct mii_info<‘a> *next;
   };
   extern int netif_start_queue(struct net_device<‘a> *dev);
   ```

   Note here how recursive fields (`mii_info.next`) are restricted to have the same type variable with the same identity as the enclosing declaration. The algorithm for introducing these type variables is a worklist style method that iterates until a fixed point is reached. Mutually recursive data structures do, however, pose a problem. This method of type variable introduction is provided as a convenience to the user to minimize the tedious manual annotation of types. For this reason, in

```
extern ``C include'' {
  typedef unsigned char u8;
  struct net_device {
    void *priv;
    struct net_device *next;
    int (*open)(struct net_device *dev);
  };
  struct mii_info {
    int phy_id;
    struct net_device *dev;
    struct mii_info *next;
  };
  extern int netif_start_queue(struct net_device *dev);
  extern void memcpy_wrapper(u8 *src, u8 *dst, unsigned int len);
  extern void* kmalloc(unsigned int);
  extern void kfree(void*);
}
cyclone_override {
  struct net_device<`a::A> {
    `a *priv;
    struct net_device<`a> *next;
    int (*open)(struct net_device<`a> *dev);
  };
  extern void memcpy_wrapper(u8 *{valueof(`n)} src,
                             u8 *{valueof(`m)} dst,
                             unsigned int len)
                             @requires((len < numelts(src)) &&
                                       (len < numelts(dst)));
}
export { * }
cyclone_hide { kmalloc, kfree }
```

Figure 1: A Cyclone Mechanism for Importing C Declarations

the case of mutual recursion, Cyclone rejects the program with a notification of the set of the types that form a cycle. The user then must manually annotate the declarations that are mutually recursive with the appropriate type variables.

In the second mode, (activated by the command line switch `--cifc-inst-tvar`) Cyclone adopts a "least-general" approach to introducing type variables. It is possible that (as illustrated in subsequent drivers) the most-general approach results in the introduction of too many type variables. In such cases, the least-general approach will instantiate each type-variabl with a default value of a suitable kind. In the case of our example, the result is as follows:

```
struct mii_info {
  int phy_id;
  struct net_device<void*> *dev;
  struct mii_info *next;
};
extern int netif_start_queue(struct net_device<void*> *dev);
```

Note that arbitrary type overrides are possible as long as they are representation consistent. For example, the type of `memcpy_override` is overridden to reflect the necessary bounds information using Cyclone's framework for pre- and post-conditions.

2. Asm expressions

   The gcc compiler supports inlined assembly instructions within C code. The linux kernel makes heavy use of this feature. Although inherently unsafe, we needed to support this feature to be able to write kernel code. The Cyclone parser and binding phase were modified to allow for gcc style assembly instructions within Cyclone.

3. Miscellany

   A number of small features that are supported by the C99 standard needed to be added to Cyclone. These included features such as compound literals, (some others too...)

4. Cyclone runtime support

A cyclone driver requires a number of runtime services. These include features such as exception handling, array bounds checking, region based memory allocation, etc. The section on the cyclone runtime driver specifies in detail the services that are available to a a cyclone driver.

1.1.4 Open Issues / Cyclone enhancements required

1. A Capability Based Allocation Mechanism

   Consider the case of the allocation of a `net_device` . The interface requires that the network device be allocated using the following function.

   ```
   struct net_device *alloc_etherdev(unsigned int sizeof_priv);
   ```

   This function allocates a net_device structure with enough space for the private data that will be stored within the net_device structure. In recent versions of the kernel (2.6+), network devices are also expected to be freed using a complementary function `free_netdev`. Thus the allocation routines for this kind of structure are beyond the control of a standalone cyclone driver – i.e, it is not possible to allocate this net_device structure as a reference counted object, since we cannot allocate the additional header word required for the reference count. Nor can we treat this object as a unique pointer, since clearly many aliases to this object are maintained (within the pci device, the mii structures, the network interface structures etc.).

   For this kind of long lived data object that admits multiple aliases, one typical Cyclone solution is to use dynamic regions. In this case, allocation within the region page assigned to the dynamic region is not possible, since the allocation itself is performed outside of Cyclone. One approach to obtaining a type-safe mechanism to support this idiom might be to use the capability system that comes with the dynamic region system, while not using the en-masse allocation scheme that is typical of regions. Such an allocation scheme might look like the following:

```
extern ``C include''
  struct net_device *`r cyc_alloc_etherdev(unsigned int privsz,
                                           region_key<`r> *`U key) {
       return alloc_etherdev(privsz);
  }
  void cyc_free_etherdev(struct net_device *`r dev,
                         region_key<`r> *`U key)
                         __attribute__((consume(1, 2))){
      free_etherdev(dev);
 }
export { * }
```

Here the standard heap-allocation mechanism of `alloc_etherdev` is augmented by a unique capability that is required to access the allocated data. Freeing the device consumes both the key and the device, thus preventing it from further use by the driver. Note that the region key here has no runtime significance at all.

2. Reading/writing non-pointer values

   The driver acquires io-memory as a pointer to an array of char (say, `unsigned char ?@nozeroterm`). This array can represents registers that reside on the device. A particular register might be a byte register, or a word, or even a double word register. Reading or writing to such registers is inconvenient in Cyclone, although it is possible to do in a typesafe manner. For instance, the following function safely reads a u16 from a u8*.

```
extern ``C include'' {
  u16* read_u16_from_u8(u8 *x, int offset, int bound) {
    if(offset + sizeof(u16) > bound)
      _throw_arraybounds();
    return *((u16*)(x+offset));
  }
}
cyclone_override {
  u16@`H read_u16_from_u8(u8*valueof(`n)@nozeroterm x,
                          int offset, int bound)
    @requires((bound == numelts(x)) && (offset < numelts(x)));
}
```

Cyclone already allows non-tagged unions as long as the only values that are ever read from the union are not of pointer type. This restriction ensures that a pointer cannot be manufactured from an integer. Instead of using this somewhat unwieldy library function, it should be possible to read and write non-pointer values of a various sizes from an fat-pointer.

3. Declarative scheme for tagged unions

   The `netdev_ioctl` function makes use of a pointer to user-space that logically is identical to a tagged-union. The first word of the memory is a tag that refers to the contents of the remainder of the memory. To support this in Cyclone required writing unsafe C code that performed the conversion between this implicit tagged union and Cyclone's tagged-union. This process is error prone and perhaps should not be exposed to the driver developer.

   An alternative approach might be to modify the Cyclone code generator such that the representation of a tagged union is specified declaratively. Such a declaration may take the following form:

```
tagged_union_layout(id=ifr_data)
  tag:(offset=0, size=4); /* specifies the location and size of the tag*/
  match tag with /*specifies the relation between tag and content*/
    ETHTOOL_NWAY_RST -> int
    ETHTOOL_GDRVINFO -> struct ethtool_drvinfo*
    ETHTOOL_GSET -> struct ethtool_cmd*
    ETHTOOL_GSTATS -> struct ethtool_stats*
    _ -> err
;
/*original C declaration augmented with an attribute*/
struct ifreq {
  ...
  char * ifr_data __attribute__((tagged_union_layout(id=ifr_data)));
  ..
};
```

   As with other type overrides, we could accumulate a library of such tagged union descriptors. This would allow the driver developer to treat such implicit tagged unions as normal tagged unions without any unsafe conversion functions.

4. Flow analysis with asm expressions

   The current support for asm expressions is only rudimentary. A first step might be to include asm expressions in the flow analysis to ensure that variables are initialized etc.

1.2 Cyclone Runtime Module

Several features of the Cyclone program language rely on runtime support. These features are summarized as follows.

1. Array Bounds Checks

   Type safety for array indexing and pointer arithmetic operations are ensured by the insertion of runtime checks. Cyclone has several mechanisms that are used for encoding the length of an array. Among those that require runtime checks are *fat pointers* and *zero-terminated pointers*. In the first case, pointers to the start and end of the array are explicitly maintained. In the second case, the last element of the array is required to be zero. Assignments through zero-terminated pointers must ensure that the terminating zero is not overwritten. Thise assurance is provided by means of runtime checks.

2. Not-null checks

   Pointers in Cyclone may be qualified as *non-null*. Obviously, assignments to such pointers must ensure that null is never written.

3. Exception Handling

   Linux kernel code does not support exceptions, even if they are of the C++ variety. As described in the previous section, care must be taken to ensure that a Cyclone exception is never reaches the context of a C function. Exceptions in Cyclone do, however, provide a natural mechanism for error-handling. The Cyclone libraries make extensive use of exceptions too (including the runtime checks described above). Thus, we allow Cyclone kernel code to use exceptions. Runtime support for exceptions involves setting handlers and, in case an exception is thrown, unwinding the stack to the context to the appropriate handler.

4. Memory Management

Cyclone provides safe memory management using several techniques. The runtime support, if any, for each of these is described below.

(a) Garbage Collection

Cyclone uses the BDW conservative collector. A GC enabled Cyclone program must be linked against this collector. While in principle it is possible for a Cyclone kernel module to use a garbage collector, this is in most cases undesirable. If a driver must use garbage collection, this GC library would be part of the cyclone runtime services.

(b) Unique Pointers

Alias-free affine pointers allow for safe manual memory management – i.e malloc and free. The support for this is based solely on a static analysis and requires no runtime support.

(c) Reference-counted Pointers

The allocator for reference counted objects reserves an additional word of header space in which to maintain the reference count. Incrementing and dropping reference counts (and free-ing the memory when the reference count reaches zero) is also achieved by means of runtime functions.

(d) Lexical and Dynamic Regions

Allocating a region page, freeing a region, and allocating individual objects within a region all require runtime support. Region pages are allocated using the `GFP_KERNEL` mode of the `kmalloc` allocator.

In some cases, drivers may wish to allocate pages using `vmalloc` instead. This method of allocation is preferred when allocating large portions of memory and results in a linear array of kernel virtual addresses. Such a mode of allocation maybe be used when, say, a video driver allocates its frame buffers. To handle such cases, the Cyclone languages now provides the following function:

```
void* rvmalloc(struct _RegionHandle *r, size_t len);
```

This function allocates the smallest multiple of 4096 greater than `len` bytes using `vmalloc`. Cyclone associates the allocated pages with the regions handle and frees the memory (using `vfree`) when the region is freed.

1.2.1 Thread Safety

There are a number of issues that make Cyclone unsafe when used in a multi-threaded setting. Most of these issues arise as a result of non-atomic multiword copies (say in the case of fat pointers, or existential types). There are also issues with aliasabilities and shared region handles.

Placing many of these issues aside for now, we permit Cyclone to be used with threads and provide support for multiple runtime stacks. Cyclone applications in user-mode user posix thread support for thread local storage to maintain. Posix-threads are not available in the kernel. To provide thread local storage we use the following scheme.

The Cyclone runtime module maintains a hash-table mapping thread-ids to stack allocated storage for that thread. When a C function calls into a Cyclone function, we expect the C function to stack allocate a record of the following type:

```
struct _tls_record {
  struct _RuntimeStack *current_frame;
  struct _xtunion_struct *exn_thrown;
  const char *exn_filename;
  int exn_lineno;
};
typedef struct _tls_record tls_record_t;
struct _tls_slot {
  unsigned int pid;
  unsigned int usage_count;
  tls_record_t *record;
};
typedef struct _tls_slot tls_slot_t;
```

Here `struct _tls_slot` represents a single entry in the hash-table mapping the thread's `pid` to a record that contains the threads stack information. This record is maintained only for the portion of the thread's lifetime that is spent in a Cyclone context. Once the Cyclone function returns to its C caller, this piece of stack allocated data can is released.

1.2.2 The cyc_runtime Module

All the runtime services described here are packaged as a kernel module. This module must be loaded prior to the loading of any other Cyclone module.

1.3 Intel i810 and friends ICH driver for Linux

1.3.1 Overview of driver behavior

- **Module initialization**

  The i810 driver is a pci enabled driver. It initializes itself by calling `pci_register_driver` directly, as opposed to the more standard `pci_module_init`. Device probing is implemented by the function `i810_probe`. The actions taken by this function are as follows.

  1. Pre-initialization

     Wakes up the device using `pci_enable_device`, and attempts to set a dma mask for future consistent dma allocation.

  2. Allocating Card and IO Memory

     Allocates an `i810_card` and grabs pci resources; i.e. pci bus start and end addresses for the ac97 codec, and for the audio card itself. If memory mapped IO is available, it grabs two further pci resources and `ioremap`'s the addresses. Most, though not all (suspicious ... figure out why) subsequent accesses of io memory perform a check to decide whether or not the memory-mapped addresses are valid. The allocated card is linked into a driver static variable that maintains a list of cards.

  3. Allocating DMA Channels

     Consistently allocates three hardware channels (`struct i810_channel`), each of which is dma based. The driver is intended to support playback only, although the card itself does recording too. The recording support in the driver is incomplete. These channel objects contain the static information regarding port numbers etc. for each hardware channel. As such, after initialization they are read only.

  4. Codec Initialization

     Initializes the `AC97` codec. This codec represents the audio control mixer. Initialization involves powering up the codec bus by writing to the io-registers grabbed from the pci resource interface previously. Space for the codec structure (`struct ac97_codec`) is allocated. The ac97 interface's `ac97_probe` function is called which initializes the codec structure appropriately. The functions

associated with the mixer (volume control etc.) are registered with the sound subsystem using `register_sound_mixer`.

5. Registering Device Functions

   An irq line is grabbed for the card. The sound device (/dev/dsp) is registered with the sound interface (`register_sound_dsp`) by passing in a table of file operations associated with the device. These operations correspond to reading, writing, opening files and ioctl for the device.

6. Failure Handling

   Failure at various points in the initialization is handled by a set of cleanup labels that are jumped to from the various failure points. Cleanup involves unmapping io-memory, releasing irq lines, freeing allocated cards, channels, etc.

- **Post Initialization – Main Functional Path**

  Playing an audio file is controlled by the functions registered as dsp functions with the sound system. The sound system uses the standard file system interface. The behavior of each the file operations in the context of this card is described below.

  1. Open

     ```
     int i810_open(struct inode *inode, file_state_t file)
     ```

     This function is the main entry point into the driver. The `inode` argument is ignored. The responsibility of this function is to set up a state object associated with the file being played. A reference to the card is obtained from the global list of cards. Each card is examined for an available channel – recall that the card supports at most three hardware channels. Once a card with a free channel is found, a state object `struct i810_state` is allocated. This object is the logical channel that corresponds to an in-use hardware channel.

     The state object is at the heart of an intricate alias graph. The state object maintains a reference to its containing `struct i810_card` object, which in turn has references to each of its open states. Recall that a ref to the card is also maintained in a global list data structure. A reference to the state is also stashed in the file object.

This function also does some clock rate tuning.

2. Write

```
ssize_t i810_write(file_state_t file,
                   char *valueof(‘i) @nozeroterm  buffer,
                   tag_t<‘i> bufsize, loff_t *ppos)
```

After the state initialization performed by the `open` function, the task of writing data to the device falls upon this function. Writing the file to the device is achieved using a dma scheme that is closely tied to the physical device.

The `struct i810_channel` object, as noted earlier, corresponds to a hardware channel. This channel is dma allocated (using consistent allocation). The exact layout of this object is important as the i810 card relies on this to access the data buffers copied to and from it. The `struct sg_item` object is a pair of addresses, the first of which is the bus address of dma allocated data buffers.

The hardware channel objects are allocated at module initialization. The bus addresses that is returned for this array of channels is recorded in the `chandma` field of the card, but the device itself is not notified of this dma address.

When a write (or read) system call occurs, data buffers need to be set up to to transfer the data to (from) the card. These data buffers are also consistently allocated dma buffers that are maintained along with the `struct i810_state` object, the software representation of a channel. A set of functions `prog_dmabuf`, `alloc_dmabuf` etc. are dedicated to the handling of these dma buffers. Once the dma buffers of an appropriate size are allocated, the bus addresses of these buffers are copied into the sg_item fields of the hardware channel objects (`i810_channel`). It is only after this step that the device is notified of the bus address of the hardware channels; i.e. the `chandma` field is written to a device register. Note that in this manner, the address of the dma data buffers is communicated to the device using a layer of indirection.

With the dma data buffers set up properly, writing to the devices is straightforward. The data to be copied to the device resides in user space and is pointer to by the `buffer` argument. This function sits in a loop and copies chunks of this buffer to the dma

data buffers. In case the buffers are full, it waits for the device to drain the buffers, until such time as the entire user space buffer has been written.

The check to ensure that the requisite space is available in the dma buffers reads registers from the device to update the `hwptr` field of `i810_state.dmabuf`. This field represents the last read point of the device.

3. Read

```
ssize_t i810_read(file_state_t file,
                  char *valueof(`i) @nozeroterm  buffer,
                  tag_t<`i> bufsize, loff_t *ppos)
```

The read function mirrors the write function almost exactly. The only difference being that data is copied *to* the user space buffer. The documentation of the driver notes that the card's recording functionality is not complete.

4. Poll

```
unsigned int i810_poll(file_state_t file,
                       struct poll_table_struct *wait)
```

The status of the file is determined solely by the amount of available read/write space. This signals whether the device is ready for reading or writing.

5. Ioctl

```
int i810_ioctl(struct inode *inode, file_state_t file,
               unsigned int cmd, int *arg)
```

As in the case of the 8139too driver, the `arg` parameter is a reference to an implicit union in user space memory. In this case the `cmd` is the tag, and the union is defined as:

```
tagged_union_layout(id=ioctl_arg)
tag:(parm=3, size=4);
match tag with
SNDCTL_DSP_GETIPTR   -> struct count_info*
SNDCTL_DSP_GETOPTR   -> struct count_info*
SNDCTL_DSP_GETISPACE -> struct audio_buf_info*
SNDCTL_DSP_GETOSPACE -> struct audio_buf_info*
_                     -> int*
;
```

The operations supported by this device range from simply getting version numbers, to tuning clock rates for the card.

6. MMap

```
int i810_mmap(file_state_t file,
              struct vm_area_struct *vma)
```

Like read and write, this function sets up the dma buffers and assigns the bus addresses to the hardware channels. Instead of proceeding to write to (or read from) these dma buffers, `i810_mmap` remaps the virtual addresses of the buffers to the area requested in the `vma` parameter.

7. Release

```
int i810_release(struct inode *inode, file_state_t file)
```

This function is responsible for cleaning up the driver state once the user space process gives up its handle to the file. The cleanup entails the following components.

- Draining Data Buffers
  The driver must wait for the device to finish copying any pending data in the dma buffers. In the case where the driver is in recording mode, no draining needs to take place. Instead, the device is signalled to stop writing to buffers and pointers to the buffer are reset.
- Deallocate DMA Buffers
  The DMA data buffers allocated during read/write/mmap must be released. Even though the buffers do not last for the entire lifetime of the driver, these buffers were consistently allocated. (It might be possible to stream allocate these buffers too.) Freeing the buffers involves releasing the pages that were marked as reserved, calling `pci_free_consistent` and clearing the bus addresses within hardware channels so that the device no longer can access the freed pages.
- Deallocate Software State
  The `struct i810_state` that was setup to handle this particular file must be released. Recall that references to this state object are proliferated among various objects.
- Releasing Hardware Channel

The hardware channel associated with this state is returned to the pool of available channels for this card.

- **Interrupt Handling**

  ```
  static void i810_channel_interrupt(struct i810_card *card)
  ```

  Interrupts are raised by the card to signal completion of copying from DMA buffers, or to signal updating the last valid index (LVI) in the dma buffer ring. In both cases the current position of the hardware pointer (which keeps track of the next position to read/write to in the dma buffer) is updated. In the case of the LVI interrupt, if the buffers are full then the recording/playback is also stopped.

- **Post Initialization – AC97 Codec**

  The i810 card conforms to the AC97 codec standard. Initialization of the `AC97` interface involves registering a codec data structure with the mixer component of the sound subsystem. The mixer interface expects the driver to set callback function pointers in the codec structure that allow the mixer to access the ac97 registers with the card. The initialization of the card does precisely this and sets pointers to the functions `ac97_set_io` and `ac97_get_io` in the codec structure.

  The AC97 interface is also modeled as a file system and as such requires the a `file_operations` to be supplied. The only functions that are relevant here are open and ioctl. Notably, the release function is missing. The `i810_open_mixdev` finds a card from the global list of card and initializes a file structure with the codec structure associated with the card. The `i810_ioctl_mixdev` function simply extracts the codec associated with the file and invokes the kernel ac97 interface's ioctl function passing the codec as an argument.

- **Shutdown**

  The `i810_remove` function is registered with the pci subsystem. The function assumes that there are no open files associated with this device. That is to say, `i810_remove` does not attempt to clean up any instances of `i810_state`. The cleanup is limited to the following:

– Releasing IO Memory

  Two kinds of IO Memory are held by the device. The first is ioremapped bus address for device resident memory. These are released using `release_region` and `iounmap`. The driver also holds consistently allocated pages for the hardware channels. The original driver does not release these channels – a likely memory leak.

– Unregistering Devices and Freeing Devices

  Both the AC97 codec and the soundcore unregistration functions are called. The `i810_card` should be removed from the device list global, although it isn't – a dangling pointer. The card is freed using `kfree` as are each of the codec structures associated with the card.

1.3.2 Cyclone safety mechanisms

1. Fat pointers for IO Memory

   Most of the IO operations of this driver consist of accesses to DMA'd memory. There are however operations that write to device registers that need to be made safe. Just as with the ethernet driver, the macros inb/outb were replaced by bounds-checked Cyclone versions.

2. Dependent Types for Critical Path IO

   Bounds checks on fat pointers are expensive. We discovered that using fat pointers on some paths, caused a distinct decrease in sound quality. In order to eliminate some bounds checks dependent types were used. The advantage of using dependent types is that a single bounds check allows the analysis to prove repeated accesses to a particular array element as being within bounds. This amortization of bounds checking helped improve sound quality.

   The particular form of IO memory array is such that the bounds are known for a particular card. Since this array is allocated using `pci_resource_start` and `pci_resource_end`, we know that the value returned will be the nearest page size multiple of the memory on the card. It should be possible to use an array with a static bound to eliminate many more of the runtime bounds checks.[2]

[2]We haven't explored this with the new vcgen bounds check elimination either.

3. Type Overrides

   We used the extern "C include" feature here with the `--cifc-inst-tvar` flag on to suppress excessivce type variable introduction. Once again, Cyclone types were assigned to C declarations using the cyclone_override feature.

4. Reference Counted Allocation

   Reference counted objects are one natural way to represent many of the data structures used in this driver. For instance, an `i810_card` object has several aliases. It is an element of the `devs` global card list; it is a member of the `state` object; it is referred to by the `pci_dev` structure; the ac97 codec structure holds a reference too. The lifetime of the object corresponds to the lifetime of the driver (it is allocated during probing, and freed when the device is removed). There are other cases in this driver of objects that have shorter lifetimes. Notably, `i810_state` objects have a lifetime that corresponds to that of a single file being played (or recorded) – a lifetime that obviously exceed a lexical scope. Furthermore, the aliases of a state object are several too : from the card object, from the file object. These considerations imply that reference counting is a natural idiom for managing the allocation of these objects.

   One port of the driver uses reference counting extensively – all instances of `struct i810_card` and `i810_state` were made reference counted. The relevant kernel data types (such as the pci device) were made region polymorphic too, and were instantiated using `'RC`, reference counted region.

   While it was possible to encode this driver using reference counted types, maintaining all the aliases in a coherent manner proved to be cumbersome. Ensuring the aliasing discipline requires the use of the swap operator. Swapping in a null reference into a critical field of an object (such as `struct i810_card.state`) is prone to failure since it temporarily destroys the alias graph on which the driver is critically dependent. Consider the following code fragment which manufactures a local alias of the state object maintained within the card object.

```
struct i810_state *‘RC temp_state = NULL;
struct i810_state *‘RC local_state = NULL;
struct i810_card *‘RC temp_card = NULL;
temp_card :=: devs;
temp_state :=: temp_card->state;
local_state = alias_refptr(temp_state);
temp_state :=: temp_card->state;
temp_card :=: devs;
```

All the data structures are restored to their original state at the end of this code fragment (with an additional alias to the state object). Code of this sort is used extensively throughout the driver. The nature of this device is such that it is frequently interrupted. While a file is being played, the device is reads data out of the dma buffers in parallel with the driver thread that writes the data into these buffers. The LVI type interrupts are raised by the device to cause the driver to update its record of current read position of the device. Since the above code must run in contexts that have interrupts enabled, and since no provision of reasonable cost (such as disabling interrupts etc.) can be made to ensure its atomicity. For this reason, we were unable to use reference counting succesfully. Interrupt handling code would often run with the pointer graph in an invalid state, resulting in unexpected null pointer exceptions.[3]

5. Dynamic Regions

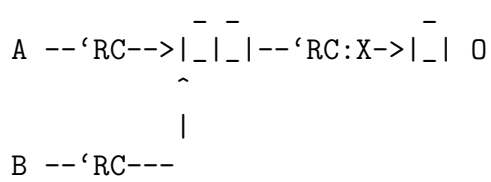
   Using dynamic regions allowed for freely aliased objects within a particular region, while still permitting the explicit deallocation of these objects. Our solution was to use a combination of existential and universal quantification, and dynamic regions to represent a state object. Cyclone disallows placing alias-restricted pointers (`‘U` and `‘RC`, i.e., those in a region of kind `TR`) directly within an existential type. The

[3]The swap operator while intended to be atomic does not currently have an atomic implementation. It is also worth noting that only aliases to the `struct i810_state` objects were found to be critical. That is, it was possible to use reference counting for the `struct i810_card` object and not have interrupt handlers fail. Finally, the `ac97_codec` object was never implemented using reference counting. It is unlikely that the aliases of this object are on a critical path, and it should be possible to use reference counting here. The point remains, however, that the non-atomic manufacture of aliases although type-safe is impractical in the presence of frequence context switching.

only way to examine the contents of an existentially quantified object is by unpacking it using pattern matching. Pattern matching would result in the illegal creation of a alias-restricted pointer. Consider the following example

```
struct wrapper {
  <‘dummy::I>
  int *‘RC a;
};
int use_wrapper_ok(struct wrapper *‘U w)
__attribute__((consume(1))){
  let &wrapper{a} = w;
  return *a;
}
int use_wrapper_fail(struct wrapper *‘r::TR w) {
  let &wrapper{a} = w;
  return *a;
}
```

Note that if pattern matching to extract the reference counted pointer from an alias-free object, then we can simply consume the wrapper to guarantee that no aliases have escaped. If, however, there are aliases to the wrapper, (either reference counted, or simple heap directed pointers), then consuming the pointer does not suffice. In the example below, consuming the pointer A after extracting the pointer `‘RC:X`, allows that pointer to be extracted once more using the pointer B. Clearly this is unsafe since the object O could have been freed using the reference `‘RC:X` thus creating a dangling pointer when unpacking B.



One way to overcome this restriction is the use of the following scheme:

```
struct state_reg <‘r::R> {
  rcregion_key_opt_t<‘r> regkey;
  struct i810_state *‘r state;
};
struct state_wrapper{<‘r>
  struct state_reg<‘r> *‘H s;
};
void handle_state(struct state_wrapper *‘RC sw) {
  let &state_wrapper{<‘r> s} = sw;
  rcregion_key_opt_t<‘r> tmp = NULL;
  s->regkey :=: tmp;
  rcregion_key_opt_t<‘r> rgn= alias_refptr(tmp);
  s->regkey :=: tmp;
  {
    region h = open(rgn);
    state->field ...
    ...
  }
  drop_refptr(rgn);
}
```

Note here that the existentially quantified `struct state_wrapper` contains only a heap pointer which can be used without a problem in pattern matching. Extracting the reference counted region key `regkey` from `struct state_reg` can be done as usual using the swap operator since it does not have to be unpacked using pattern matching.

We used this scheme to represent the software state of the driver. The advantage of this scheme over reference counting is that within the existentially quantified dynamic region objects of type `struct i810_state` can be freely aliased. The function `handle_state` above shows how the number of swaps required to manipulate the state is greatly reduced – within the scope of the region declaration no swaps are needed. We found this approach to be much more reliable than standard reference counting.

The glaring disadvantage with this method is that it leaks memory. We require the contents of the existentially quantified object to contain a heap directed pointer, so that the aliasing discipline is observed. Such pointers cannot however be freed. Thus, with each allocation of a state

object, we leak two double-words of memory corresponding to the size of `struct state_reg`.

1.3.3 Open Issues / Cyclone enhancements required

An Alternative Construct for Unpacking Existential Types

A prominent failing of the Cyclone port of this driver is the memory leak due to the interaction between existential types and alias-restricted pointers. We attempted a solution to this problem using a new construct for pattern matching. This construct would appear as follows:

```
struct ex_type {
  <`i::I>
  tag_t<`i> a;
  int *{valueof(`i)} `RC b;
};
void bind_address(struct ex_type ex) {
  let ex_adr<`i> = ex;
  let btmp = NULL;
  btmp :=: ex_adr->b;
  ...
}
```

Here the `let<`i>` construct binds the existential type variable of the `struct ex_type` and binds thee *address* of `ex` to the variable `ex_adr` which has type `struct ex_type *`bind_address`. By explicitly specifying the scope of the type variable, this scheme avoids the well-known difficulty of inferring the scope. Also, by binding the address it provides a more convenient syntax for unpacking an existential, and would have allowed for swapping out pointers to obey the aliasing discipline.

Unfortunately, we learnt that this method of unpacking is unsound. Dan Grossman's ESOP paper (ref?) describes similar unsoundnesses due to the interaction between unpacking existential types and the address-of operator. An illustration of the unsoundness is provided below.

```
struct Closure {
  <‘a>
  void (@f)(int, ‘a);
  ‘a env;
};
void use_closure(struct Closure @‘H c, struct Closure @‘H c2) {
  let p<‘a> = c; //bind struct Closure @p=c, ‘a as witness type of p
  let f = p->f; //f has type void(@)(int, ‘a)
  *c = *c2; //changes the witness type ‘a
  f(0, p->env); //oops -- unsound
}
```

One option was to permit this kind of unpacking only in the case where the pointed to object is declared as `const`. However, since C allows the introduction of the `const` qualifier simply by using a cast, guaranteeing the const-ness of the underlying data structure is not feasible. Furthermore, our solution for using `TR` kinded pointers within these existential types requires the swapping out of these pointers, which clearly violates const-ness.

1.4 PWC driver for Philips webcam

The pwc driver is intended to support multiple usb cameras using the video4linux interface. Although primarily intended for Philips models, it also supports models by Logitech, Samsung etc. Our experiments were conducted using a Logitech QuickCam Pro 4000 usb camera.

It is worth noting that this driver is not supported in linux kernel distributions after 2.4.28 (?). The reason for this appears to be th reliance on a binary only release of an image codec since it is Philips proprietary code. Using this binary only code required a special hook to be installed within the kernel. The provision for this hook was removed in later distros of the kernel. As a result, the entire pwc driver has been discontinued. Presumably alternative open source drivers will become available soon. The version of pwc used here does not utilize this codec at all and as a result is purely at the sourc level, i.e. no binary only code.

1.4.1 Overview of driver behavior

- **Module initialization** This driver is intended for devices on the Universal Serial Bus (USB) rather than on the PCI bus. It follows that

the setup for this driver differs substantially from the previous drivers described here. The interface provided by the kernel usb service does, however, bear resemblance to the pci interface.

`static int usb_pwc_init(void)`

This function is declared as the module initialization routine. It relies on the static initialization of `video_device` and `usb_driver` structures. Similar to pci device structures, these static objects maintain reference callback functions within the driver that the kernel invokes to perform various operations on the device.

Module initialization parameters, if any, are parsed and the appropriate driver globals are set. The module parameters allow for the setting of the image and frame sizes, the number of buffers used for collecting images, the number of frames per second, image compression, power saving, led control, a debugging flag, and finally a hint parameter that might help the driver recognize the device.

Once all globals have been set properly, the `usb_register` function is called, passing the `usb_driver` structure as an argument. This results in `usb_pwc_probe` function being called, which performs the real work of initialization. Specifically, when this function is called, the existing list of interfaces is rescanned so that this driver can attach to any recognized device already present. The driver's probe functions is also called when a device is plugged into the usb port.

```
static void *usb_pwc_probe(struct usb_device *udev,
                           unsigned int inum,
                           const struct usb_device_id *id)
```

The attached device is recognized using a tuple of vendor-id and product-id that is maintained within the `udev` structure. A `pwc_device` structure, the principal object in this driver, is allocated. This structure maintains the various structures associated with image collection, together with values that relate to the video device and usb device interfaces. The basic values related to frame size etc. are set in this structure.

The next step involves the allocation of a video device structure. As mentioned previously, the driver must make use of the video4linux facility (/dev/video), which is provided by the video driver module. It is the

responsibility of this driver to allocate and initialize a `video_driver` object which, similar to the `usb_driver` object, defines various references to driver callback functions. Some of these fields can be set statically, while others rely on the identity of the actual device probed. In addition, the `video_driver` object allows the driver to stash a reference to its private data within it, which in this case is the `pwc_device` object, which also maintains a reference to the `video_driver` object. This double linkage impacts our choice of a memory management solution for this driver.

Once the `video_driver` object is set up, the `video_register_device` function is called. The role of this function is primarily to assign a node number to this device (typcically `/dev/video0`), and binds the supplied `video_device` to an array of video_device objects currently registered on the system.

Failure Handling????

- **Post Initialization – Main Functional Path**

  An application that attempts to access the video device must first call the `pwc_video_open`, among the functions that had been registered with the video4linux system previously. The set of such functions mirrors the file operations as registered by the audio driver.

  1. Open

     ```
     int pwc_video_open(struct video_device *vdev, int mode);
     ```

     The main function of this operation is to set up the buffers used by the driver to gather images from the camera. After the camera is powered-up, these buffers are allocated. There are three levels of buffers:

     (a) Isochronous Buffers
     These buffers are used to gather data that is received in small packets (urb's) from the usb controller. The number of these buffers is fixed statically to 2. Each isoc buffer has a fixed size and is simply allocated using `kmalloc GFP_KERNEL`.
     Isoc buffers are closely coupled with the urbs used by the usb interface. Urb's must be allocated using `usb_alloc_urb`

passing in the number of packets that each urb contains. This allocator allocates the urb structure followed by enough space for packet descriptors for the number of packets requested per urb frame. It remains the responsibility of the usere to allocate the data buffer in which these packets will be stored. To this end, the iso buffers allocated using `kmalloc` are stored in the urb. Once the urb structure has been initialized, it is registered with the usb system usinig `usb_submit_urb`. When all the packets in an urb have been filled and interrupt is raised and the driver copies the packets in the urb into the current frame buffer.

It is to be noted that the size of the frame buffer and the size of an urb frame are vastly different. That is, several urbs are required to fill a single frame. Frame boundaries are marked by a packet of distinguished size.

(b) Frame Buffers

Data gathered form the ISOC buffers are assembled into frame buffers. The maximum number of frames is fixed statically to 5. The frames are organized in a circular list which keeps track of empty frames and frames that are filled and awaiting hand-off to the user process. The size of each frame buffer is large and may span multiple pages. These pages are allocated using `vmalloc`.

(c) Image Buffers

Image buffers are mmap'd buffers that are used to copy data from the frame buffers into memory that is accessible by the user space process. In case there is image compression being used then the image buffer contains the decompressed data. The image buffer data is embedded within a viewport too.

Image buffers are allocated using `vmalloc` which is suitable for grabbing multiple pages of memory that is not required to be physically contiguous. In addition to vmalloc'ing the pages, the image buffers are also mmap'd to an address requested by the user space process so that they can be easily handed off.

2. Interrupt

```
static void pwc_isoc_handler(urb_t urb)
```

When setting up the urb structure during `video_open`, this function is registered within the urb as the interrupt handler for processing completed urbs. This function accesses the current partially filled frame and proceeds to copy data from the urb's transfer buffer into this fill-frame.

If the size of an urb packet is below some recognized threshold for this device then this is intended to signal the end of the frame. In this case, after copying over the packet to the fill-frame, the isoc_handler wakes up any read process that might be waiting on a shared queue for a frame to become full.

3. Read

```
long pwc_video_read(struct video_device *vdev, char *buf,
                    unsigned long count, int noblock);
```

This function is invoked by the user process when a image frame is required. In most cases, however, this process must block until such time as a frame becomes available. That is, partial reads from a frame are typically disallowed. For this purpose, in the initialization code, a wait queue is declared for processes to wait for completed frames. Upon entry into this function, if it is the case that there is no completed frame waiting to be read, then this thread will wait on this queue until it is woken up by the `pwc_isoc_handler` urb completion interrupt handler. Note that this function does not sit in a loop waiting each time for a frame to become available. It only handles a single frame, and once that has been copied to the user the function exits.

Without decompression handling frames is relatively straightforward. The current frame pointers are advanced so that the isochronous process can begin to fill in new urb packets into frame buffers. The data from the filled frame buffer is copied over to the current image buffer. This copying requires a step of conversion from the camera's native format to a yuv format, and also includes the creation of a viewport (border frame) in the image buffer. Finally, once the data is copied from the image buffer to the user space buffer pointed to by the `buf` argument.

4. MMap

```
int pwc_video_mmap(struct video_device *vdev,
                   const char *adr, unsigned long size);
```

The driver provides this facility to map all the image buffers to the requested memory address starting at `adr`. However the symbols that refer to the current read positions and valid frame buffers etc. are not exported by this driver. Thus, the calling process must manage reading data out of the image buffers by itself. This provides a mechanism for partial reads from these image buffers.

5. Poll

```
unsigned int pwc_video_poll(struct video_device *vdev,
                            struct file *file,
                            poll_table *wait);
```

This function returns a status that indicates whether or not there are frames available for reading.

6. Ioctl

```
int pwc_video_ioctl(struct video_device *vdev,
                    unsigned int cmd, void *arg);
```

7. Close

```
void pwc_video_close(struct video_device *vdev);
```

This function releases all buffers associated with the video device. Freeing the buffers is done in the opposite order of allocation. That is, first image buffers are released, then frame buffers, then isoc buffers and urbs. Note that urb's must be freed using the `usb_free_urb` interface.

This function does not free any other structures associated with the drivers such as the `video_driver` and `usb_driver`. That responsibility falls upon the usb interface's callback functions.

- **Shutdown**

1. Usb Disconnect

```
static void usb_pwc_disconnect(struct usb_device *udev, void *ptr)
```

This function is not part of the v4l interface. It is registered with the usb subsystem along with the `probe` function during driver initialization. As the name suggests, this function is called when the camera is unplugged from the usb port. It is at this point that cleanup operations are performed.

Allocation and freeing of the `video_device` structure is the responsibility of this driver. As noted in the description of `usb_pwc_probe` the driver allocated the video device and then registers it with the v4l subsystem. Thus, the cleanup operations performed here must free this object to prevent a memory leak.

However, nothing prevents the user from disconnecting the cable while some v4l application is still in use. In this situation, the `video_device` structure is still in use in the v4l drivers and freeing this object will result in a dangling pointer in the v4l drivers. To avoid this situation, the driver makes the following choice: A count of the number of open video devices is maintained in the `pwc_device` object. Each call to `pwc_video_open` results in this count being incremented; calls to `pwc_video_close` decrement the count. If the cable is disconnected while this count is greater than zero the driver delays freeing the object. Instead, a reference to this soon-to-be obsolete object is stashed in a driver global variable. All other state associated with this device is discarded. The driver anticipates that a subsequent call to `pwc_video_read` by the v4l application will fail with an error code that is an indication for the application to exit. Once the v4l application has exited it is now safe to discard the stashed `video_device` object and reclaim the memory leak. Thus, an additional responsibility of `usb_pwc_probe`, `usb_pwc_disconnect`, `usb_pwc_exit` is to free any pending object that is stashed in this global variable.

In the normal situation (the video open count is zero), the driver goes ahead and frees the video device object immediately. In both cases the driver unregisters the device with the v4l subsystem.

Finally, this cleanup routine also deallocates the main `pwc_device` object.

2. Module exit

```
static void usb_pwc_exit(void)
```

The counterpart of `usb_pwc_init`, this function is called when the module is finally unloaded. It handles any `video_device` memory leak that may have been generated by a premature usb cable disconnect. It also deregisters the driver from the usb subsystem.

1.4.2 Cyclone Safety Mechanisms

1. Dynamic Regions for pwc_device object

   As in the case of th `i810_state` object in the previous driver, the `pwc_device` object was represented using an existentially quanitified heap directed pointer to a region key and the actual pwc object itself. As previously, this representation has the advantage of permitting free aliasing of the pwc object within the region, while still allowing the object to be freed manually. There remains the problem of the quadword memory leak.

2. Image buffers

   The three layers of image buffers are used by this driver to encode the state of a single user application. With each session of video_open, video_close these buffers are allocated and freed.

   The ISOC buffers were allocated in the reference counted region. These buffers are only referred to through the urb and through the isoc buffers themselves. They do not participate in any more complicated structure that makes reference counting unwieldy.

   The frame buffers are maintained in two circular lists, one for the empty frames and one for the filled frames. In addition, the current position in each list is maintained as well. This aliasing pattern makes using reference counting troublesome. As a result, much as was the case with the pwc_device object, frame buffers were allocated in dynamic regions. Once again, this encoding results in a small memory leak.

   The image buffers are also maintained in a circular list. These were also allocated in dynamic regions. Both frame and image buffers are allocated using the `rvmalloc` construct, as described in the section on the cyclone runtime module.

3. Exception Checking

As mentioned previously, exceptions that are thrown from Cyclone code must not cause the stack to be unwinded into C code. One construct described earlier was the use of the `cyc_call` macros that ensure that a top level handler is installed when a C function calls into a Cyclone function. While effective, this mechanism is unwieldy in that often a surrogate function is required for each cyclone function that may be called from C. That is, for a Cyclone function `foo` we require a C function `C_foo` whose body consists only of a `cyc_call` to `foo`. The function `foo` is hidden from the C namespace and only `C_foo` is exposed. Clearly, this duplication is cumbersome. More importantly, this approach does not lend itself easily to checking that no exceptions escape Cyclone.

Cyclone function types can be annotated with an attribute `no_throw`[4]. Such functions must have bodies that consist mainly of a try/catch block in which a top level handler is installed. Some code that is guaranteed to not throw exception (such as simple variable declarations) are permitted outside this try-block.

We annotate the type of the various function pointers that are installed in kernel datastructures (such as the video_driver callback functions) with this `no_throw` attribute. Thus we use the type checking system to ensure that no exceptions leak into C code.

```
struct video_device<‘a::A>
{
  ...
  int (*open)(struct video_device<‘a> *, int mode)__attribute__((no_throw)
  void (*close)(struct video_device<‘a> *)__attribute__((no_throw));
  long (*read)(struct video_device<‘a> *, char *, unsigned long, int noblo
  ...
  ‘a *priv;
  ...
}
```

4. Handling urb packet descriptors

   Allocation of urbs must be done using the `usb_alloc_urb` interface. As described previiously, this function takes an integer argument that

[4]Although not described in each case, this mechanism was used for all but the ethernet driver described here

specifies the number of packets that are contained within a single urb. This function contiguously allocates enough space for the urb itself followed by space for descriptor fields for each of the packets. The definition of an urb then is as follows:

```
struct urb {
 ...
 int npackets;
 ...
 struct iso_packet_descriptor desc[0]; //Note the zero array dim.
}
```

The definition of an urb is closely coupled with the implementation of the usb modules. It is not possible to override the declaration of this structure such that it is explicitly an dependent type with the `npackets` field specifying the number of elements in the `desc` array. Such an override is not representation consistent since it uses a non-zero array dimension which changes the size of the urb object.

To overcome this we used an escape into C that explicitly constructs a dependent type using the address of the desc field. This utility function is packaged with the other usb type overrides. A Cyclone driver is expected to use this interface to access the packet descriptors.

```
struct iso_packet_desc_array {
  <`i::I>
  tag_t<`i> len;
  struct iso_packet_descriptor *{valueof(`i)} base;
};

typedef struct iso_packet_desc_array iso_arr_t;

extern "C include" {
  iso_arr_t build_iso_arr(struct urb *u) {
    iso_arr_t ret = {u->npackets, &u->desc};
    return ret;
  }
}
```

1.4.3 Open Issues / Cyclone enhancements required

1. Urb allocation/de-allocation – Type based restrict construct

   Urb's are explicitly allocated and deallocated using an interface that is outside the control of the cyclone driver. One approach to ensuring the safety of this kind of memory management is to use the separate capability technique as described in section **??**.

   An alternative approach is to use a combination of reference-counted (or dynamic regions) and unique pointers to support this idiom. This method requires a couple of steps. First, the allocation function (`usb_alloc_urb`) is hidden from the Cyclone program (using the `cyc_hide` construct of extern "C include"). Next a wrapper to this function is provided, and the type of the deallocator is overridden as follows.

```
struct urb_wrapper {
  struct urb *‘U urb;
};

struct urb_wrapper *‘RC cyc_usb_alloc_urb(int npackets);

void usb_free_urb(struct urb *‘r u) __attribute__((consume(1)));
```

   The cyclone application that wishes to manipulate several aliases of the urb object can do so using the wrapper. The requirement of swapping out the inner urb object before accessing (due to the unique path requirement on unique pointers) ensures that the aliasing discipline is maintained. By consuming its argument, the type of the deallocator effectively encodes the fact that it frees its argument.

   A downside to this approach is also the need to swap out the unique pointer each time. As was the case with `struct i810_state` in the sound driver, this kind of destructive manipulation of a shared data structure can be hard to use. One approach that was considered to remedy this situation, was to use a type-based restrict operation. In effect, restrict-ing on a particular pointer value would prevent dereferencing all pointers that are a subtype of the restricted pointer's type within the scope of the restrict operation. Of course, the restrict-ed pointer itself would be allowed to be dereferenced. This operation would allow

for the elimination of the swap construct, preserving the integrity of the data structure. In the multi-threaded case however, some more complex construct would be required.

2. Video device allocation/deallocation

   The `video_device` object is allocated and deallocated by this driver, although a large proportion of the functionality related to this object is managed seperately in the v4l drivers. As described previously, the newly allocated object is handed off to the v4l subsystem using a registration protocol. Thus, care must be taken when deallocating the object, since references to it might still be present from the v4l modules.

   One approach to managing this might be to allocate the `video_device` object using a reference-counted region. An additional attribute might be used on a function type to specify that the function manufactures an alias to this pointer. That is, we might have the following types:

```
void video_register_device(struct video_device *‘RC dev)
     __attribute__((inc_refcnt(1)));

void video_unregister_device(struct video_device *‘RC dev)
     __attribute__((dec_refcnt(1)));
```

   These types would be used by the code generator to insert the appropriate reference-count changes when the function is called. In this manner, all the references within the v4l subsystem are treated as a single reference. This reference counting mechanism could also be used to replace the application level reference counting used by the driver to manage memory leaks in the case of usb cable disconnects.

   Reference counting, as experienced previously, is difficult to use, especially in the case of an object that is widely aliased even within this driver. In such cases the preferred option is to use dynamic regions. A similar mechanism might be used to handle this situation, by using the following wrappers.

```
void cyc_video_register_device(struct video_device *‘r dev,
                               region_key_t<‘r> *‘RC key )
```

```
    __attribute__((inc_refcnt(2)));

void cyc_video_unregister_device(struct video_device *‘r dev,
                                 region_key_t<‘r> *‘RC key )
    __attribute__((dec_refcnt(2)));
```