Uniform Driver Interface

UDI Environment Implementer's Guide Version 1.0, First Edition

http://www.project-UDI.org/specs.html

UDI Environment Implementer's Guide

Abstract

The UDI Environment Implementer's Guide is an informative document that provides guidelines and suggestions for implementer's of UDI environments based on the following UDI specifications:

- UDI Core Specification, Version 1.0
- UDI Physical I/O Specification, Version 1.0
- UDI PCI Bus Binding Specification, Version 1.0
- UDI NIC Driver Specification, Version 1.0
- UDI SCSI Driver Specification, Version 1.0

This material is not to be considered part of, or replacement for any part of, the corresponding specifications. The intended audience for this book includes environment implementers.

See the Document Organization chapter in the UDI Core Specification for a description of the UDI Specification collection, as well as references to additional informative materials.

Status of This Document

This document has been reviewed by Project UDI Members and other interested parties and is believed to provide accurate tutorial information. It contains informative material only, and is not to be cited as a normative reference from another document. Implementations are not required to comply with the suggestions in this document.

Copyright Notice

Table of Contents

Environment Implementer’s Guide *1*

1.1 Overview

The Uniform Driver Interface (UDI) specifications define a complete environment for the development of device drivers. This includes a driver architectural model and the complete set of services and other interfaces needed by a device driver to control its device or pseudo-device, and to interact properly with the rest of the system in which it operates. This environment in which a driver operates, called the UDI environment, must be provided in each operating system or operating environment in which UDI drivers operate.

This document describes how to implement a UDI environment, and is thus targeted towards system vendors who provide the driver services and other aspects of the UDI environment in a system.

All implementations of UDI require a set of code that converts OS specific services and requests into UDI services and requests. This is the code, for example, that turns a UDI request for memory into a real request to the underlying OS for memory, dealing with any blocking issues that the underlying OS might impose.

There are many different issues and architectural concerns that have to be handled by the environment code. Many of these issues will be discussed in this section and different solutions and constraints will be presented. It is the responsibility of the environment implementer to weight these different solutions and concerns and choose a solution that is appropriate for the particular OS that the environment will support.

The UDI Environment Implementer’s Guide is organized into the following main sections:

Include Files	Discusses issues with the implementation of the UDI include files that UDI environments provide for drivers.
Error Handling	Discusses how UDI environments can deal with misbehaved drivers.
Resource Usage	Discusses general issues and opportunities associated with the managing of resources in the UDI environment
Control Block Management	Discusses issues with the management of control blocks in UDI environments.
Memory Management	Discusses issues with dynamically allocated memory in UDI environments.
Buffer Management	Describes the management of UDI buffer objects in UDI environments.
Time Management	Discusses the issues with the implementation of timers and timestamp interfaces in UDI environments.
Instance Attributes	Discusses issues related to the implementation of UDI instance attributes.

Inter-Module Communication	Describes the creation and management of channels and regions.
Tracing & Logging	Discusses issues related to the implementation of tracing and logging services.
Debugging	Discusses issues related to the implementation of debugging services.
Utility Functions	Discusses issues related to the implementation of utility functions.
DMA	Discusses issues related to the implementation of DMA services in UDI environments.
PIO	Discusses issues related to the implementation of PIO services in UDI environments.
Interrupts	Discusses issues related to interrupt handling.

1.2 Include Files

The UDI specifications define include files that are provided by environments and included by drivers. For example, all of the interfaces in the UDI Core Specification are obtained by including the file "udi.h"; the interfaces in the Physical I/O Specification via "udi_physio.h"; the SCSI Metalanguage interfaces via "udi_scsi.h"; etc.

Only udi.h and udi_physio.h contain definitions that have environment-dependent components; all other include files (for metalanguages or bus bindings) are completely portable and need not vary from one environment to another, and could even be provided by a third party vendor. Therefore, we'll focus on the implementation of udi.h and udi_physio.h in this section. See the Reference Implementation for details.

Environment implementers may choose to structure these header files as two monolithic files or divide them up into smaller pieces, as is done in the UDI reference implementation. In either case, the difference must not be visible to device drivers.

Aside from structural issues, the only environment-dependent components of these include files are (1) the definitions of UDI's fundamental data types, and (2) the definitions of implementation-dependent macros.

1.2.1 Definitions of Fundamental Types

The fundamental data types in UDI are composed of specific-length types (types that are guaranteed to be of a specific size), abstract types (types that are sized appropriately for a given class of environment implementation), and opaque types (handles and udi_timestamp_t). The specific-length and abstract types are integral types which need to be typedef'd to the appropriate basic C types (char, short, int, etc.). The opaque types need to be typedef'd appropriately for the needs of the given platforms and relative to a given ABI definition, if applicable.

Because of the variability in size and alignment of the basic C types, it is strongly recommended that UDI environment code use UDI's fundamental types for its own data structures, rather than the basic C types whenever possible to enhance portability and simplify support of multi-platform environments.

1.2.1.1 Specific-Length Types

The definitions of the specific-length types depend on the sizes of the underlying basic C types. Fortunately, the following C language definitions will produce the correct results with all three programming models of most interest on IA-32 and IA-64 systems (ILP32, LP64, and P64). In all of these programming models, the basic types char, short, and int, are 8, 16, and 32 bits respectively; the long and pointer sizes vary, but are not needed to implement UDI specific-length types.

```
typedef     unsigned char     udi_ubit8_t;
typedef     char              udi_sbit8_t;
typedef     unsigned short    udi_ubit16_t;
typedef     short             udi_sbit16_t;
typedef     unsigned int      udi_ubit32_t;
typedef     int               udi_sbit32_t;
typedef     udi_ubit8_t       udi_boolean_t;
```

1.2.1.2 Abstract Types

Abstract types can in general vary in size from one environment to another. However, UDIG-compliant systems will conform to the IA-32 or IA-64 ABI binding, which specifies the sizes of the abstract data types: udi_index_t is 8 bits on both IA-32 and IA-64, and udi_size_t is 32 bits on IA-32 and 64 bits on IA-64. The following definitions can be used with ILP32 and LP64 programming models:

```
typedef     unsigned long                       udi_size_t;
typedef     udi_ubit8_t                         udi_index_t;
```

For a P64 compiler that uses “long long” to express 64-bit integral types, udi_size_t would have to be defined as:

```
typedef     unsigned long longudi_size_t;
```

1.2.1.3 Opaque Types

Opaque types can vary in both size and structure from one environment to another. The IA-32 and IA-64 ABI bindings define the size of all opaque types to be equal to the size of a pointer (32 bits on IA-32, 64 bits on IA-64), so a simple implementation of the opaque type definitions is the following:

```
typedef void *_UDI_HANDLE;
typedef     _UDI_HANDLE udi_buf_t;
typedef     _UDI_HANDLE udi_channel_t;
typedef     _UDI_HANDLE udi_constraints_t;
typedef     _UDI_HANDLE udi_timestamp_t;
typedef     _UDI_HANDLE udi_pio_handle_t;
typedef     _UDI_HANDLE udi_dma_handle_t;
typedef     unsigned long     udi_timestamp_t;
```

For a P64 compiler that uses “long long” to express 64-bit integral types, udi_timestamp_t would have to be defined as:

```
typedef        unsigned long longudi_timestamp_t;
```

Handles are opaque references to environment objects so a simple implementation of a handle is to use a void * cast to a pointer to the object. However, other handle implementation choices may make more sense for certain environments, such as untrusting environments that want to protect the environment object from being corrupted by a misbehaving environment.

1.2.2 Implementation-Dependent Macros

Given the choice of using void * for handles, the two handle-related implementation-dependent macros can be defined trivially as:

```
#define UDI_HANDLE_ID(handle, handle_type)(handle)
#define UDI_HANDLE_IS_NULL(handle, handle_type) \
                                    ((handle) == NULL)
```

1.3 Error Handling

This section deals with errors encountered in the UDI interfaces such as when a driver makes a UDI service request with an illegal parameter or when a driver dereferences an invalid pointer. Normal device errors, such as a disk time out, are expected to be dealt with by the associated drivers. The OS dependent environment has multiple choices for how to deal with errors in the UDI interfaces.

1.3.1 Ignore the error

A very simplistic implementation can decide to just ignore these errors and continue execution. Since this represents a very serious failure within the UDI sub-system ignoring these errors most likely will ultimately result in an OS crash. The faulting UDI code will continue and may eventually do something that corrupts the entire OS. Ignoring UDI errors is not a recommended technique, unless the driver is believed to have been previously tested in a more rigorous environment.

1.3.2 Terminate region

Since all UDI driver code runs inside the context of a region a better technique is to just terminate the region that made the illegal request. This would enable any other instances of the driver to continue and would only penalize the driver instance that contained the error.

Note that all resources that have been allocated for a region need to be reclaimed when the region is terminated in this fashion. The UDI environment needs to ensure that all dynamically allocated resources are linked in such a way that they can be. Note that in the case of abnormal termination the environment also needs to be able to reclaim partially allocated resources.

NOTE: Before abruptly terminating a driver instance, the environment must stop the corresponding device from issuing more interrupts or DMA cycles. It does this by executing the PIO abort sequence transaction list previously registered by the driver via `udi_pio_abort_sequence`.

1.4 Resource Usage

The goal of any UDI implementation is to avoid any static, predefined limits on the size and number of resources used by the implementation. However, in practice there can be situations where a given implementation is forced to pre-allocate certain resources. The following is a discussion of those resources that might create such problems.

1.4.1 Interrupt vectors

In order to deliver the appropriate context to a UDI interrupt handler, the UDI environment needs to be able to associate multiple context pointer values with a given interrupt vector so that each handl emay be called with its associated context pointer. If the native OS does not provide this capability directly then the UDI environment will have to simulate this capability, possibly by creating an array of internal data structures, one for each interrupt vector. This array of structures can either be allocated dynamically or statically, depending upon how the OS enumerates the interrupts available in the system.

1.4.2 Allocation daemon

As described in Section 3.4.6.4 there is an alloc daemon whose job it is to handle memory allocation request that have to be delayed for some reason. This implies that there will be a queue of allocation requests that will be queued up for the daemon. Since the UDI common code creates a queue of requests for the alloc daemon using a linked list, there should be no fixed size issues associated with this daemon but this is an area to consider carefully.

1.4.3 Metalanguages and Mappers

UDI metalanguage libraries, which contain the code for metalanguage-specific channel operations, need to be loaded and initialized prior to the drivers and external mappers that use them. The external mappers may also need to be initialized before the associated hardware device drivers can be started.

For OS's that suppoer dynamic loading of system modules, this is simply an issue of establishing the correct inter-module dependencies so that the various modules load in the right order.

On those systems without dynamic loading, the metalanguages and mappers will have to be initialized statically, either by a set of function calls coded directly into the UDI environment initialization code or by calls based upon a static table of supported metalanguages and mappers.

1.4.4 UDI device drivers

The actual UDI device drivers also need to be statically initialized for operating systems that don't support dynamic driver loading. The difference is that these drivers normally cannot be initialized from the UDI initialization code. Since drivers control actual hardware they cannot be started until the OS is ready to deal with devices, which frequently occurs after the UDI system itself is initialized. For this reason initialization of static UDI drivers will sometimes require changes to the underlying OS itself, unless a "glue" layer can be attached to the UDI driver to make it appear to the OS as a native driver.

Again, there is no problem with an OS that supports dynamic driver loading: the UDI driver is installed and initialized when needed.

1.4.5 Region identification

One of the basic concepts of the UDI architecture is the region. The important capability is that the UDI machine specific code frequently has to identify the region associated with a specific device. You can do this by creating a fixed size array of pointers to regions and then associating that array with the devices in an OS specific manner. This is an acceptable technique as long as the array of pointers is sized appropriately and this does not create an excessive amount of overhead since the array only contains pointers to regions and not the region structures themselves.

Another way to solve this problem is to add a region pointer to the control structures that the OS uses to identify devices. This is a more general technique but requires more invasive changes to the underlying OS and therefore may not be acceptable. The specific implementation must decide which of these two techniques is more appropriate.

1.5 Control Block Management

A control block contains context data that is associated with any request that goes to or from a UDI driver. You need to consider the following issues when dealing with these control blocks.

1.5.1 Scratch space layout

Associated with a control block is some scratch space that contains temporary data for a driver. The contents of the scratch space are not guaranteed to be preserved when the control block is passed between regions. Also, the receiving end of a request may have specified a scratch space size that differs from, and may be larger than, the amount of scratch space currently associated with a control block. There are some different techniques that can be used to deal with the scratch space.

1.5.1.1 Inlined scratch space

The implementation is free to define a fixed amount of scratch space that is part of the control block itself. If this space is large enough then no further scratch space needs to be allocated. If a request comes in that needs more scratch space than is available in the inlined area then a dynamic area can simply be allocated for that control block.

1.5.1.2 Dynamic allocation

A simple technique is to just have the receiving end of a request free up the scratch area and allocate a new area of the appropriate size. This technique suffers from the problems that it will slow down the UDI system by adding many more memory allocation requests and could also lead to deadlock situations. Nevertheless, in practice, a steady state should be reached quickly as the scratch space for a particular control block grows to the maximum size used with that control block.

1.5.1.3 Allocate maximal size

Another possible technique is to allocate a scratch area that is large enough for any driver request and attach that area to the control block when the control block is allocated. This solves the performance and deadlock problems at the expense of allocating more memory than is needed for every control block in the system.

The trick is determining the maximum size, especially in the presence of dynamically loaded UDI modules, since there is no a priori way to determine the maximal scratch size utilized by these modules.

Since there is an upper bound of 4000 bytes on scratch space size, some implementations may choose to pre-allocate 4000 bytes of scratch space and keep this permanently associated with the control block. However, this technique should be used with care, as it will consume a large amount of memory. It is most likely to be useful when control blocks are otherwise allocated in separate MMU pages, such as, for example, when isolating drivers in separate address spaces.

1.5.1.4 Multiple scratch area pools

Pools of scratch areas of different sizes can be maintained. Whenever a control block needs a scratch area of a different size the current scratch area is returned to it's pool and a new scratch area from the appropriate pool is obtained. This technique solves the performance problem but still has some potential deadlock issues.

The problem is what to do when there are no more appropriately sized scratch areas. The problem can be delayed by allocating a scratch area from the next larger size pool but eventually all of the pools could be empty, at which point in time the system will have to allocate new memory for the scratch area pools, potentially leading to deadlock.

1.5.2 Marshalling area

If an implementation requires extra memory for marshalling data from the sender of a control block request to the receiver then similar issues as those encountered for scratch areas will arise. The amount of marshalling space needed is a function of the number and size of the parameters to a given request and completely determined by the associated metalanguage. If the implementation only supports a pre-defined set of metalanguages, sizing the marshalling area is a simple matter of finding the metalanguage request that needs the biggest marshalling space and then using that for all control blocks.

The problem is a little more difficult if the UDI implementation supports metalanguages that can be loaded at run time. In this case, there is no way to know, a priori, how big the marshalling area must be. In this case, the environment can just allocate marshalling space for the largest request that it knows about before hand and then provide one of the dynamic techniques described for scratch areas that will handle the obscure requests that are bigger than expected.

1.6 Memory Management

One of the fundamental attributes of the UDI system is that all service calls, including memory requests, follow a non-blocking execution model. To implement this on top of a more traditional memory allocation service, the memory allocation routines in the UDI reference implementation achieve this model by using a two step process to allocate more memory. First the UDI code will make an optimistic non-blocking allocation request for more memory. This request will immediately return, either successfully returning the requested memory or an indication of failure. If the allocation succeeds then the UDI code calls the callback routine using an "immediate" callback, passing it a pointer to the allocated memory.

If the allocation fails then the memory request cannot be satisfied at this time. The UDI code then adds the request to a queue and lets an allocation daemon deal with the request. The allocation daemon, since it is running in a separate context, is free to make a blocking request, which may block in order to satisfy the allocation request. When the allocation daemon ultimately succeeds in satisfying the request, the daemon will call the callback routine, passing a pointer to the allocated memory.

1.6.1 Non-blocking memory allocation requests

These requests provide instant gratification and are guaranteed to return immediately, whether or not the memory for the request is available. If the memory is not available the request just returns a null pointer, indicating the allocation failure. If the underlying OS does not provide a non-blocking memory allocation routine then the UDI implementation will have to simulate this somehow, either by checking for the availability of memory before making the actual allocation request, allocating memory out of a pre-defined area, or by always failing atomic allocation requests. The latter choice, however, will likely have adverse performance impacts. The important point to stress here is that, no matter how the implementation achieves it, this request must not block.

1.6.2 Blocking memory allocation requests

A blocking memory allocation request is one for which it is acceptable for the calling thread to be blocked in order to satisfy the request. One aspect of this type of allocation request is that it will never return failure. Either this request will return a pointer to the desired memory, potentially sometime in the future after the calling thread was blocked and then resumed, or this request will never return. If the underlying OS's blocking memory allocation request has the potential of returning failure then wrapper code must be provided that checks for this condition and reissues the request until the request is satisfied. Note that just reissuing the request is probably not sufficient; the wrapper code will undoubtedly have to call some OS services that free up memory or wake up a paging daemon or do something that will cause memory to become available.

1.6.3 Zero sized requests

Some code in the UDI environment reference implementation may make zero-length allocation requests to the underlying OS-dependent allocation routine. That allocation routine must return a NULL pointer in this case without raising any kind of error indication. For those OS's that consider a zero size allocation request an error some wrapper code will have to be provided that checks for this situation and returns the NULL pointer without ever calling the OS's allocation routine.

1.6.4 Allocation daemon

The UDI system is based on the concept of callbacks. Any service request that could block is passed a pointer to a callback function that is called when the request is satisfied, potentially after the request blocked and then later unblocked. This is central to the solution to the problem of what to do when a memory allocation request cannot be immediately satisfied. Normally UDI services will make an atomic memory allocation request. If the request is satisfied then the UDI service will complete the request and call the callback routine. If the memory allocation request fails then the UDI service will queue the request so that it can be handled later by the Allocation Daemon.

The Allocation Daemon is an OS thread that blocks waiting for memory allocation requests to be placed upon its queue. When it finds such a request on its queue the Allocation Daemon dequeues the request and then makes a blocking memory allocation request to the OS. When the memory allocation request returns the Allocation Daemon calls the associated callback routine and then looks for more work.

There are multiple different ways the allocation daemon may be configured. The UDI implementation is free to use any implementation that is most appropriate to the underlying operating system. Some obvious possibilities are:

1.6.4.1 Single daemon

This is the simplest implementation and consists of one allocation daemon for the entire system. All allocation requests are placed on one queue that is serviced by one daemon. The daemon grabs the first request on the queue and makes a blocking request for memory to satisfy that allocation request. When the memory is allocated the daemon enables the region that made the request and calls it's callback routine, completing the request.

While simple, this technique has the attribute that all allocation requests are blocked waiting for the first request to complete. Once the allocation daemon issues a memory request all other allocation requests on its queue must wait until that memory request completes. A single request for a large amount of memory might block other requests for smaller amounts of memory that could be satisfied immediately.

1.6.4.2 Daemon per CPU

On a multi-processor system it might be more efficient to create a separate allocation daemon for each CPU in the system. With this scheme an allocation request is placed upon the queue for the daemon that runs on the current CPU. This way there is no need to switch contexts from one CPU to another, both when making the allocation request and when completing the request. This will potentially result in better TLB utilization and faster context switch times.

1.6.4.3 Daemon per request size

Another technique is to create multiple daemons, each one of which is responsible for allocation requests of a specific size. This way small sized requests would not have to block waiting for a large size request to complete. It is also possible to have a specific daemon take advantage of any simplifications that the native OS provides. As an example many OS's provide a separate, highly optimized interface for allocating page-sized memory. By having two different allocation daemons, one for page sized requests and a different one for all other requests, it becomes easy to take advantage of such interfaces.

1.7 Buffer Management

It is important to note that UDI drivers are presented with a logical view of data buffers and not an actual physical area of memory. As such the OS dependent environment is responsible for managing and controlling the actual layout and access to buffers and needs to deal with the following issues.

1.7.1 Linked list of areas

If physically contiguous memory is difficult to obtain then a better technique for dealing with buffers would be to represent them as a linked list of physically contiguous areas, each separate area being no larger than a page size. The operating system dependent environment is responsible for creating and maintaining the control structures required to mange these separate areas.

A critical decision that needs to be made is whether to use a static pool of these control structures or whether to dynamically allocate them as needed. Dynamically allocating these control structures avoids artificially limiting the number and size of buffers that can be allocated due to a lack of control structures. The problem with dynamic allocation is that this could add a memory allocation request to each I/O request. Since a memory allocation request could block this could significantly slow down I/O requests and could lead to a deadlock situation. If a disk write request needed to allocate a buffer control structure at the same time that the system needed to issue a disk write request in order to free up memory so that memory requests could be satisfied the system would suddenly be deadlocked.

This area requires careful engineering analysis to decide where static control structures are required and where dynamically allocated control structures can be tolerated.

1.7.2 Native OS Buffers

For many I/O requests, the native OS must provide a buffer that request must operate on. The best technique is for the UDI environment to use this buffer directly when an I/O request needs a buffer. This can be accomplished by modifying the internal UDI buffer management data structures to add enough information to keep track of the native buffer and then use the native buffer directly. The biggest problem in this area is the case of a set of discontiguous buffers that are used for DMA transfers. UDI uses the IEEE 1212.1 format to describe the segments that comprise an I/O block vector. If the native OS uses the same format then there is no problem using the native buffer when an I/O block vector is needed. If not then the UDI environmental code will have to allocate the IEEE 1212.1 data structures needed to describe the native buffer and fill these data structures in appropriately.

1.7.3 Valid versus allocated buffer size

A UDI buffer has two sizes that are associated with it, allocated size and valid size. The allocated size represents the total amount of space that has been allocated for the buffer. The valid size represents the total amount of space that contains actual information. Buffers are frequently passed around in the UDI system and valid size changes. Rather than constantly re-allocating buffer space and copying valid data from the old buffer to the new one the environment can just keep track of the allocated versus valid size of a buffer. If the buffer needs to change to a smaller size then the buffer is kept and just the valid size is changed. If the buffer needs to be larger than its allocated size then a new buffer must be obtained but only the valid data needs to be copied to the new buffer, thereby cutting down on the amount of memory to memory copying required.

1.7.4 Copy on write

A buffer can be copied from one buffer to another. The obvious way to do this is to simply copy the data. A potentially more efficient way of doing the copy is to use copy on write. The memory management structures for the destination buffer are set up to point to the source buffer but they only provide read only access. If the destination buffer is never written then the copy is accomplished with no memory copy. If the destination buffer is written, the kernel has to take a page fault, allocate new memory for the destination buffer, and then copy data from the source buffer to the destination buffer.

Note that this technique requires more complex data structures to keep track of which memory pages belong to which buffers. Also, this technique only works if the native OS accepts page faults while running in kernel mode. One downside to this technique is that buffers that ultimately need to be copied will require an extra page fault, add a certain amount of overhead to the copy operation.

1.8 Time Management

[TBS]

1.9 Inter-Module Communication

1.9.1 Synchronization

The region represents the basic synchronization paradigm for UDI. Calls into a region are serialized with respect to calls into the same region and data is not shared between two different regions. Multiple regions, however, even in the same driver instance, can execute concurrently.

Consideration must be given to how the different regions are synchronized from each other. The environment must weigh carefully the advantages and disadvantages of using spinlocks, sleeplocks or queues to achieve this synchronization.

Spinlocks should only be used in situations where a small amount of code must be synchronized. When large blocks of code need to be synchronized then a sleeplock or a queued daemon would be more appropriate. A case in point is the PIO engine. The number of PIO operations that can be created for one request is unbounded and therefore synchronizing the PIO engine with a spinlock could result in long spin delays. A better approach would be to use a sleeplock in this situation so that useful work can be done while waiting for a long PIO engine request to complete.

The queued daemon technique can be utilized by the memory allocation code. With this approach, a single daemon extracts and executes requests that are placed upon a queue. Since only one daemon is running at any point in time, synchronization is guaranteed. This technique works because the daemon can only remove requests from the list and the requesters can only add requests to the list.

Only the specific implementation can determine which technique is more efficient: sleeping while waiting for a lock or queueing requests up for a separate daemon.

1.10 Tracing and Logging

[TBS]

1.11 Debugging

[TBS]

1.12 Utility Functions

[TBS]

1.13 DMA

1.13.1 Scatter/Gather lists

The UDI system uses the scatter/gather list format that is defined by the IEEE 1212.1 standard. As such, if the host OS uses the same scatter/gather list format, translating to a UDI scatter/gather list for DMA buffers becomes trivial. You just use the scatter/gather list provided by the host.

If the host operating system uses a different format, the OS dependent environment has to translate between the two formats. The issue that arises from this scenario is where to find space for the translated list. Allocating new memory for every DMA request creates performance and deadlock problems. Since DMA requests normally come from mapper modules then the mapper code is the place to solve most of these problems.

One solution is to have the mapper code maintain a pool of scatter/gather lists. When a request comes in with a DMA area the mapper code can allocate one of the scatter/gather lists from the pool and translate the host scatter/gather list into the pool list.

Another technique is to have the mapper code maintain private data for every possible request that could be received. Part of this private data would be a scatter/gather list that would then contain the translated scatter/gather list from the host operating system. All UDI code would then use this translated scatter/gather list.

Note that both of these solutions only work if there is an known upper bound on the size of the scatter/gather list that the host operating system provides. If there is no known upper bound then some sort of dynamic allocation scheme will have to be provided to account for very large scatter/gather lists.

1.14 PIO

Programmed I/O (PIO) operations are I/O instructions issued by driver that transfer data initiated by CPU to access registers or memory on a device. Following are issues that the implementors need to consider for implementing PIO.

* PIO transaction descriptor (`udi_pio_trans_t`) vs. PIO trans list

* Serialization of trans list

* Temporary registers

* Endianess for temporary registers

* Synchronization operations

1.14.1 udi_pio_trans

A PIO transaction descriptor contains data that describes a single PIO register/memory access transaction. Such transaction descriptors can be combined into a single list called a trans list which can perform a series of PIO transactions with a single call to `udi_pio_trans`. The implementation allows the transactions in a trans list to perform various different operations such as bit manipulation, repeat counts, and serialization. The implementation can pre-compute the transfer size of a trans list, pre-analyze or pre-scan the list, and/or compute branch targets for optimization. The environment can also scan the list for errors to prevent data corruption or system panics.

An extreme alternative would be to precompile the entire PIO trans list to native machine instructions at `udi_pio_map` time, and then call that code directly from `udi_pio_trans` (via an indirect function pointer stored in the PIO handle), instead of interpreting the trans list.

1.14.2 Serializing trans list execution

The implementation can serialize PIO transactions by using either spin locks or queuing mechanisms. However, since the execution time for trans list is unbounded, spin locks are not recommended; queuing mechanisms should be employed instead.

In addition, the environment implementation needs to choose the granularity of serialization to use. The finest granularity allowed is at the level of a PIO register set. Other choices include the entire device, the entire system, or somewhere in between. Choosing finer granularities will allow a certain amount of increased parallelism, but since the time spent for each trans list should be quite small, the benefit is limited. For a general-purpose OS, device granularity is recommended.

1.14.3 Temporary Registers

During execution of `udi_pio_trans` the environment maintains a set of 8 temporary registers, designated as register 0 through 7. Each of these registers can hold one item of up to 32 bytes (256 bits) in size. Most PIO operations use one or more of these registers. Therefore, to optimize, the environment should do operations based upon trans list size, or based on native word size. If the size of a value loaded into a register is less than 32 bytes the most significant bytes are marked all zeroes.

Implementations may choose to store PIO temporary registers on the stack or in memory (to save stack space). If they are stored in memory, they must be stored at a granularity not coarser than the chosen PIO serialization granularity. In fact, since there’s no benefit to storing registers at a finer granularity than the serialization, it is recommended that the locks or queues and the registers be stored in the same place. For example, if the serialization is at the whole device level, the register memory should be part of the device structure.

1.14.4 Endianness for temporary registers

Temporary registers represented in LE or BE will have effects on different platforms supporting different endianness architectures. Some platforms provide endian swapping in hardware. The values loaded from or stored to a device will be device specific.

Endianness of the internal register set may also be controlled by whether the implementer wants to do operations, when possible, on native words. Similarly, the size of the native word chosen may depend on the availability of the ability to manipulate them efficiently across all ops. For example, manipulating the temporary registers internally as 'ints' could be a problem when doing addition if you don't have an internal type that can handle the potential carry cases.

Be careful when implementing UDI_PIO_LOAD_IMM: each 16 bits of immediate value from the ***`operand`*** member of a trans list entry is in the driver's native endianness, but the first trans list operand always holds the two least significant bytes, the next one holds the next least significant two bytes, and so on.

1.14.5 Synchronization operations

UDI spec defines three synchronization operations for PIO:

1. UDI_PIO_BARRIER
2. UDI_PIO_SYNC
3. UDI_PIO_SYNC_OUT

UDI_PIO_BARRIER guarantees that any PIO transactions prior to UDI_PIO_BARRIER are made visible to the device when the call is complete. Any subsequent PIO transactions will not be executed before UDI_PIO_BARRIER is completed. UDI_PIO_SYNC and UDI_PIO_SYNC_OUT operations provide an even stronger synchronization that the will wait for the effect of the PIO transactions to reach the device. UDI_PIO_SYNC affects all PIO transactions, and UDI_PIO_SYNC_OUT is required to only affect output operations to the device.

The environment should also prevent panics in the event of a fault, i.e. access a device which may not exist. With non-self identifying devices like EISA enumeration needs the `udi_probe` functions to enumerate the device.

1.15 Interrupts

1.15.1 Shared interrupts

The UDI system does handle shared interrupts as long as the host OS supports shared interrupts. The one problem is that the UDI system passes an opaque data pointer to its interrupt handling code. If the underlying OS's interrupt system does not allow for an opaque data pointer for each interrupt handler then the OS dependent environment must simulate this capability. The easiest way to do this is to create an array of data structures for each interrupt in the system and utilize a dummy interrupt handler. This data structure will contain the actual UDI interrupt handler that needs to be called and the opaque data pointer associated with that handler. The dummy interrupt handler will then use the data structure to call the real UDI interrupt handler, passing it the appropriate opaque data pointer.

1.15.2 Interrupt region synchronization

Some drivers will use interrupt regions for handling interrupts; others will use interrupt preprocessing. Interrupt regions may need to be protected from preemption by device interrupts on some systems that use level-sensitive interrupts. The easiest way to do this is to just disable all interrupts when entering the

UDI interrupt region, guaranteeing appropriate synchronization between all interrupt regions. Although simple this is a fairly brute force technique. A better technique is to just disable interrupts from the device associated with the region (and other devices sharing the same interrupt line). This provides sufficient synchronization for UDI interrupt regions.

1.16 Reference Implementation

An implementation of the UDI environment services and sample UDI drivers will be provided by Project UDI. This open source code implementation will be released publicly to enable quick spread of the UDI technology to a wide range of platforms and drivers. It is expected to initially provide support for most of the major UNIX vendors as well as Linux.

This reference implementation has been co-developed by a number of OSVs and IHVs with this goal – easing the spread of the UDI technology – in mind, and the result is an implementation of the UDI environment in which much of the environment code is common across the range of supported OSes. For example, UDI's buffer management services are coded as a common piece of source code with OS dependencies abstracted via a handful of OS-dependent macros. Thus, to get these buffer management services implemented for a new OS all that the OS implementor has to do is to fill in the handful of macros to map to the needs of that OS. Some OSes may require additional work, depending on how closely the OS's kernel services map to the reference implementation assumptions, but in any case this will be a significant aid to new development.

The reference implementation is a tool that will be used by the industry at large in the development of UDI environments and drivers. See the Project UDI web page at `http://www.project-UDI.org` for additional information.

1.17 Driver Development Kits

UDI Driver Development Kits (DDKs) are expected to be available from various OSVs, which will provide UDI build and runtime environments, including driver packaging and build tools, and other driver development materials, such as test suites and driver writer's guides. See the Project UDI web page at `http://www.project-UDI.org` for additional information.

This web page also includes introductory and tutorial materials useful for UDI device driver writers.