Intel® 3100 Chipset Enhanced DMA Driver for Linux

User's Manual

Rev 1.0

March 2006

Reference Number: 310859-001

The Intel® 3100 Chipset Enhanced DMA Driver for Linux may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Contents

Figures

Tables

Revision History

Date	Revision	Description
January 2006	001	Initial release on FDBL.

§§

1.0 Product Features

1.1 General Features for Linux Kernel Version 2.4.21

- Implemented in C language and intended for use as an Open Source module.
- Compliant with the GNU Public License (GPL).
- Tested and written for these Linux kernel versions as a point release:
  - 2.4.21-4.ELsmp (WS)
  - 2.4.21-27.ELsmp (AS)
- Implemented as a driver-to-driver model with a client driver API.
- Thread-safe/Re-entrant.
- Supports the following features of the Intel® 3100 Chipset enhanced Direct Memory Access (EDMA) Controller:
  - Memory to memory (MM) and memory to memory mapped I/O (MMIO) transfers between two physical addresses with a 36-bit address range for both MM and MMIO transfers.
  - Maximum transfer size of 16 MBytes per block.
  - Chain mode EDMA transfer with four independent channels.
  - Programmable independent alignment between source and destination addresses.
  - Increment of the source and destination address.
  - Increment of the destination address and decrement of the source address to enable byte stream reversal.
  - Constant address mode for the destination address based on the transfer granularity to enable targeting of memory mapped I/O FIFO devices.
  - Buffer/memory initialization mode.

§§

2.0 Overview

This document describes the installation and general usage details of Intel® 3100 Chipset Enhanced DMA Driver for Linux; henceforth referred to as the EDMA driver. The EDMATerminology

Item	Description
Intel® 3100 Chipset Enhanced DMA Driver for Linux	The enhanced DMA driver whose features and API are described in this document. Also referred to in this document as the “EDMA driver.”
Client Driver API	An API present in the EDMA driver and exposed to potential client drivers.
EDMA_W	Refers to the Intel® 3100 Chipset platform (also a build directive).
Intermodule Communication	Process by which modules can use a Linux interface to share data (functions, strings, variables) between modules.
EDMA FIFO Mode	A constant address memory mode for the destination memory address in the EDMA_W platform.

2.1 System Requirements

All EDMA driver development and testing was performed using the Red Hat* Enterprise Linux AS operating system version 3.0. While the code is portable to other versions, no testing has been performed on other kernel versions or with other versions of the GNU C Compiler. The driver was built successfully with the Intel® Compiler for Linux (v. 8.1) on the 2.4.21-4.ELsmp kernel; however, the resulting binary was not tested extensively or validated. Any specific kernel version porting is left to the user. It is not necessary to build the kernel, but the kernel source must be installed to build the driver. The table below describes the necessary system requirements.

Table 1. System Requirements for Building with Kernel Version 2.4.21

System Requirement	Version	Notes
Red Hat Enterprise Linux WS or AS	Version 3	
Linux Kernel Source Code	Version 2.4.21-4.ELsmp Version 2.4.21-27.ELsmp (AS or WS version)	Update 5 was applied to the original AS and WS kernels. Applying an update will change the version number.
GNU C Compiler (gcc)	Version 3.2.3	Available on Red Hat Enterprise Linux Version 3 distribution.

The EDMA driver is available as a gzipped tarball (edma_driver-1.0.tar.gz) for the Linux environment. The EDMA driver source code may be extracted to any directory of preference.

Example code extraction on a Linux machine:

```
[user@machine] mkdir /mydir/edma; cd /mydir/edma

[user@machine] cp edma_driver-1.0.tar.gz /mydir/edma/

[user@machine] tar -zxf edma_driver-1.0.tar.gz
```

After code extraction, the following files will be in the working directory:

Table 2. Driver File List

File Name	Description of Contents
dma.c	The exposed client driver API, functions internal to the driver, undocumented test API for application-style unit testing.
dma.h	Public data structure definitions, constants, register offsets, string messages, ioctl values, etc.
dma_internals.h	Data structure definitions internal to the driver.
dma_list.h	API for managing lists internal to the driver.
dma_api.h	List of function prototypes in the Client Driver API.
Makefile	Commands for building driver.
ReleaseNotes.txt	High level summary of building the driver for Linux 2.4.21 .
310859.pdf	This document.

2.2 Build and Installation

The build and installation instructions assume that the Linux kernel source code is stored in the /usr/src directory for 2.4.21 builds. The soft links, /usr/src/linux and /usr/src/linux-2.4, should be pointing to the kernel source code directory.

Example:

```
[user@machine] ls -l /usr/src

linux--> /usr/src/linux-2.4.21-4.ELsmp

linux-2.4--> linux-2.4.21-4.ELsmp
```

If the kernel source was installed in a different location, modify the INCLUDE variable in the EDMA driver makefile or the soft links to point to the appropriate location.

Example INCLUDE variable in the Makefile:

```
INCLUDE = -I. -I/usr/src/linux-2.4/include
```

The EDMA driver is designed to be built for a specific hardware platform, the Intel® 3100 Chipset chipset, indicated by the variable EDMA_W in the makefile. The makefile also contains a variable near the top of the file called KERNEL_VERSION. The variable will be prefixed by the '#' or comment character. To uncomment the line for building, remove the '#' character in front of KERNEL_VERSION. Use the all_2_4 target to build the driver.

Example makefile contents:

```
#Choose a Kernel Version by 'uncommenting' the appropriate kernel

#version line below.

#KERNEL_VERSION = -DDMA_2_4
```

Example build for kernel 2.4.21 and the EDMA_W chipset:

```
[user@machine] cd /mydir/edma

[user@machine] make all_2_4
```

After the build has completed, install the resulting object file, dma.o, with the insmod program.

```
[user@machine] insmod dma.o
```

2.3 Build Settings and Configuration Options

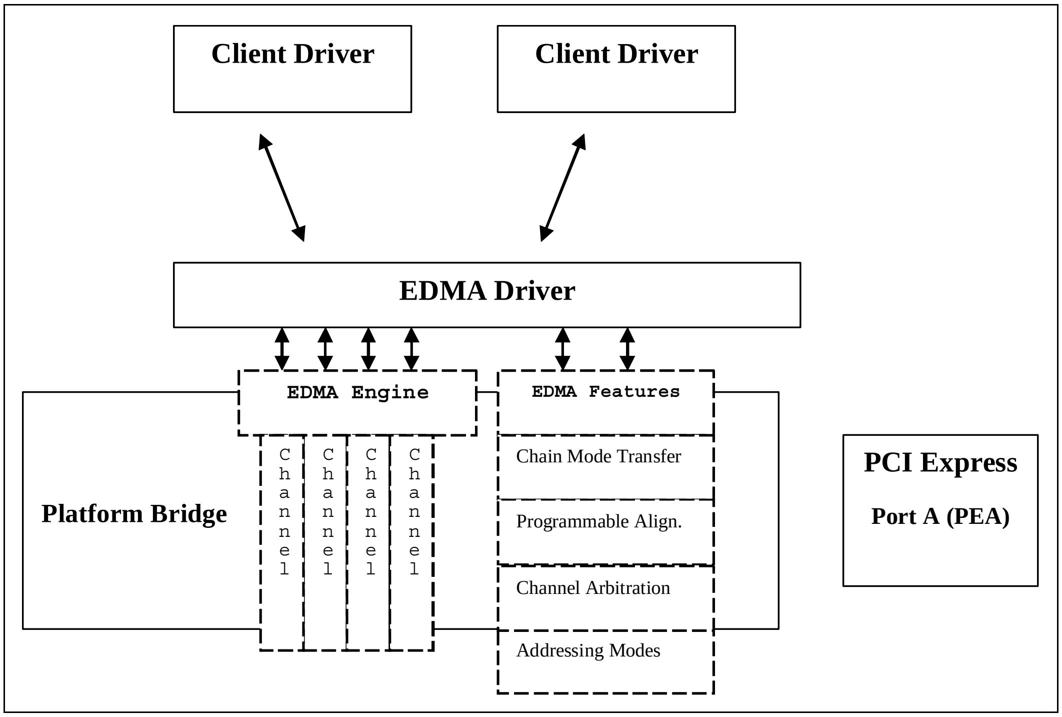
Item	Description
36-bit Addressing	Because the EDMA driver must accommodate 36-bit addressing, the EDMA driver uses the kernel type **dma_addr_t**, a type that can be 32 or 64 bits in length. The definition of the dma_addr_t type can be found in the kernel source directory in include/asm/types.h. CONFIG_HIGHMEM64G must be set for dma_addr_t to be a 64-bit type.
Channel Priority	The EDMA driver does provide the option of modifying the priority of a single channel to receive favorable latency and bandwidth service when there are competing channels. The high priority channel receives 50 percent of the bandwidth and the remaining bandwidth is distributed evenly among the competing channels. When the driver is loaded using the “insmod” or “modprobe” programs, the option of modifying channel priority is available through the use of the configuration option **edma_chan_priority**. Integer values of 0-3 are valid channel numbers. Any other value will reset arbitration to the default “round robin” scheme. Example: `[user@machine] insmod edma.o edma_chan_priority=1`
BIOS Settings	Check the BIOS to determine if the EDMA controller settings are present. At the time of this document’s creation, the BIOS was still in development. The BIOS may contain chipset configuration settings that will enable or disable the EDMA controller.
Character Driver Registration	The EDMA driver uses the register_chrdev function to obtain the major number: **register_chrdev**(unsigned int major, const char * name, struct file_operations * fops) The function dynamically selects a major number if the value of major is 0. In this case, a script needs to be created to repeatedly discover the major number and create a device node under the /dev directory with the mknod utility. In the EDMA reference driver, a non-zero major number value was used. When the major number is constant, the mknod utility is used only once to create the device node. This is not a good practice to use for long term driver development as dynamic major number allocation is the preferred method. For more information about character driver registration, see Linux documentation suggestions in the Appendix.

§§

3.0 Client Driver API

The EDMA driver is designed to be employed by client drivers in a driver-to-driver usage model. In this model, client drivers access the EDMA functionality through the EDMA driver's exposed API. The client driver API provides access to such EDMA features as channel arbitration and programmable alignment of source and destination memory addresses.

Figure 1. Driver-to-Driver Model

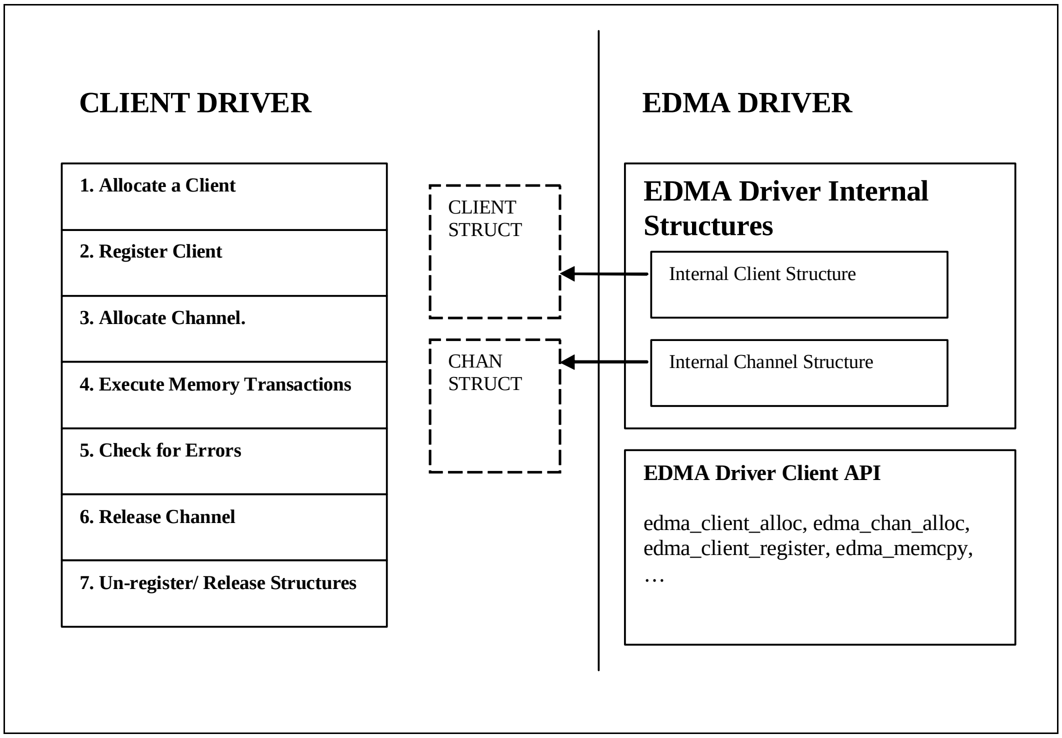


Client drivers allocate a client data structure and register with the EDMA driver as represented in Figure 2 below. A client driver fills in certain data fields in the client structure such as the "callback" function pointer which allows the EDMA driver to alert the client driver of the status of memory transactions.

Each client can obtain one or more channels; a channel can be allocated to only one client. After registration, the client driver requests the use of a particular DMA channel. If the channel is available, the EDMA driver allocates a channel structure and returns it to the client driver. Once channel allocation has completed successfully, the client driver requests memory transactions.

If the client driver has supplied the EDMA driver with a non-NULL "callback" function pointer upon registration of the client, the EDMA driver will alert the client driver when memory transactions have completed. Alternatively, the client driver may supply a NULL pointer to the EDMA driver and use the API to poll the EDMA driver to discover the status of a particular transaction.

Figure 2. Client Driver API Usage


CLIENT DRIVER
EDMA DRIVER
1. Allocate a Client
2. Register Client
3. Allocate Channel.
4. Execute Memory Transactions
5. Check for Errors
6. Release Channel
7. Un-register/ Release Structures
CLIENT STRUCT
CHAN STRUCT
EDMA Driver Internal Structures
Internal Client Structure
Internal Channel Structure
EDMA Driver Client API
edma_client_alloc, edma_chan_alloc, edma_client_register, edma_memcpy, …

3.1 Obtaining the Client API

The driver-to-driver model requires intermodule communication. To use the EDMA driver's client API, a client driver must obtain pointer references to the functions of the API. Linux provides a mechanism for intermodule communication through an interface. The sections below describe how to obtain pointer references to the client API.

The client API outlined in the table below can be found in the dma_api.h file.

Table 3. API Function Prototypes (Sheet 1 of 2)

Function Name	Description
edma_client_alloc	Allocates a client structure to be placed in the EDMA driver's client struct list and provides a client driver with a pointer to a client struct.
edma_client_release	Releases both the internal client struct and client driver's pointer reference to a client.
edma_chan_alloc	Allocates one of the four channel resources in the EDMA driver and provides the client driver with a pointer to a channel struct.
edma_chan_release	Releases channel struct.
edma_memcpy	Executes EDMA memory transactions.
edma_register	Adds allocated client struct to the internal list of clients.

Table 3. API Function Prototypes (Sheet 2 of 2)

Function Name	Description
edma_unregister	Removes allocated client struct from the internal list of clients.
edma_is_complete	Obtains progress information for a particular memory transaction. Primarily used to poll the EDMA driver to determine if a memory transaction has completed.
edma_get_errors	Obtains error information for a particular memory transaction.

3.2 API Pointer References in Linux Kernel v. 2.4.21

To obtain pointers to the client API functions provided by the EDMA driver, use the intermodule communication interface available in Linux 2.4.x kernels. The retrieval function of the client driver API has been registered using the inter_module_register function, which associates a given string with a module and data. The data may be obtained using the inter_module_get function.

```
const void * inter_module_get(const char * string);
```

Pointer references to the client driver API functions are stored in a table structure of type edma_api_export which may be obtained using the API retrieval function edma_get_functions. In the example below the inter_module_get function is called using the string “edma_get_functions.” The data that is returned is a pointer to the function edma_get_functions. After the pointer is returned, the function can be used to obtain the client driver API in an edma_api_export struct. In the example below, the edma_api_export struct variable is called dma_funcs. After edma_api_export struct is initialized, it may be used to call any member of the clientr driver API.

The names of the client driver functions can be found in Table 3 above.

Example Usage:

```
int (*edma_get_functions)();

int error_func = 0;

struct edma_client * kern_client = NULL;

static struct edma_api_export dma_funcs = { };

//Getting the function pointer edma_get_functions

edma_get_functions = inter_module_get("edma_get_functions");

//Initializing the edma_api_export struct

error_func = (*edma_get_functions)(&dma_funcs);

//Calling the client allocation function...

kern_client = dma_funcs.edma_client_alloc();
```

For more information about intermodule communication, consult the Linux documentation referenced in the Appendix. The source code examples that follow assume edma_api_export initialization similar to the one illustrated in the example above.

3.3 Allocation and Registration

Each client can obtain one or more channels; a channel can be allocated to only one client. This behavior can be modified by making alterations to the data structures and allocation functions in the driver, which are detailed in the following sections.

3.3.1 edma_client Structure

The **edma_client** structure contains fields that a client must fill out before registration with the EDMA driver as the structure is used to uniquely identify a client. The structure must be allocated and released using the client driver API as there are additional structures allocated which are internal to the driver operation.

```
struct edma_client {

    struct pci_dev *pdev;

     unsigned int chan_no; //Only 0,1,2,3

    edma_callback_t edma_callback;

};
```

Table 4. edma_client Parameters

Parameter	Description
*pdev	Pointer to the kernel structure representing a PCI I/O device. May be set to NULL for pure software clients that only need to use chipset DMA resources.
chan_no	Channel number the client is requesting.
edma_callback	Function pointer for the DMA interrupt callback, used when a DMA channel generates a completion interrupt (end of chain) or an error interrupt. Required for all clients that use DMA resources. A value of NULL will indicate that the client chooses to use the polling function to determine when a transaction has completed.

3.3.2 edma_chan Structure

The **edma_chan** structure is used internally by the EDMA driver. The structure returned from allocation is not intended to be modified by client drivers; however, the structure details are included below for reference. Client drivers are expected to treat pointers to these structures as opaque handles for DMA resources.

```
/*

 * STRUCT: dma_chan -

 * @channel_no: channels are organized by number 0...3.

 * @reg_offset: private reserved.

 * @sw_in_use: value set when a channel is allocated.

 * @dma_resource_internal: internal resource tracking mechanism

 * @spinlock_t desc_lock: for locking memory
 * @list_start free_desc: list of free descriptors available
 * @list_start used_desc: list of used descriptors available
 * @dma_cookie_t completed_cookie: last cookie completed
 * @dma_cookie_t cookie: 'current' cookie
 * @dma_cookie_t error_cookie: last cookie completed with an error
*/
struct dma_chan
{
    unsigned int channel_no;
    unsigned int reg_offset;
    int sw_in_use;
    struct dma_resource_internal resource;

    spinlock_t desc_lock;

    struct list_start free_desc;
    struct list_start used_desc;

    dma_cookie_t completed_cookie; //last cookie completed
    dma_cookie_t cookie;           //'current' cookie
    dma_cookie_t error_cookie;     //last cookie completed with err
 }
```

3.3.3 edma_client_alloc and edma_client_release Functions

The **edma_client_alloc** function allocates a new client structure, along with some additional memory allocations internal to the EDMA driver.

```
struct edma_client *edma_client_alloc(void);
```

Return Values:

- A pointer to a newly allocated structure on success.
- NULL if the structure could not be allocated due to a memory allocation error.

Example Usage:

```
struct edma_client * client = NULL;
/ * Obtain a client */
client = dma_funcs.edma_client_alloc();
```

```
/ * report error */

if(client == NULL)

{

    printk(KERN_INFO"Problem allocating client\n");

}

else

{

    dma_funcs.edma_client_release(client);

}
```

The **edma_client_release** function simply releases the allocated client structure along with some internal allocations made by the edma_client_alloc function.

```
void edma_client_release(struct edma_client *client)
```

Table 5. edma_client_release Parameters

Parameter	Description
*client	Pointer to a client instance structure, with the required fields populated.

3.3.4 edma_client_register and edma_client_unregister Functions

The **edma_client_register** function registers the filled-in client structure with the EDMA driver. Registration is important as it gives the driver a way to determine if transaction queries will be handled by polling or callback functions.

```
int edma_client_register(struct edma_client *client);
```

Table 6. edma_client_register and edma_client_unregister Parameters

Parameter	Description
*client	Pointer to a client instance structure, with the required fields populated.

Return Values:

- 0 on success
- -1 on failure

The example below illustrates typical usage of the allocation and registration functions. An edma_client is allocated and its member elements are initialized.

Example Usage:

```
/* A function to test the client callback feature */

void test_client_event_callback(struct edma_client * client, struct edma_chan *
chan, edma_cookie_t cookie, enum edma_status_t status, unsigned long user_data);

struct edma_client * client = NULL;
```

```
client = dma_funcs.edma_client_alloc();


/ * report error */
if(client == NULL)
{
   printk(KERN_INFO "Problem allocating client\n");
}
else
{
     /* Channel that the Client will be requesting. */
     client->chan_no = 0;
     client->pdev  = NULL;
     /*
      * Initialize the callback pointer
      * A value of NULL here would indicate polling is the
      * preferred method for querying transactions.
      */
     client->edma_callback = test_client_event_callback;
     /* Register, then unregister*/
     if(dma_funcs.edma_client_register(client) == 0)
     {
        dma_funcs.edma_client_unregister(client);
     }
```

3.3.5 edma_chan_alloc Function

The client driver must allocate an EDMA engine channel resource to be used for the memory transactions using **edma_chan_alloc**. If the channel requested is available, a non-NULL value will be returned.

```
struct edma_chan * edma_chan_alloc(struct edma_client *client);
```

Table 7. edma_chan_alloc Parameters

Parameter	Description
*client	Pointer to a client instance structure, with the required fields populated.

Return Values:

- NULL on failure
- Non-NULL value on success

Example Usage:

```
void test_client_event_callback(struct edma_client * client, struct edma_chan *
chan, edma_cookie_t cookie, enum edma_status_t status, unsigned long user_data);


struct edma_client * client = NULL;

client = dma_funcs.edma_client_alloc();

struct edma_chan * chan = NULL;


/ * report error */

if(client == NULL)

{

    printk(KERN_INFO"Problem allocating client\n");

}

else

{

     /* Channel that the Client will be requesting. */

     client->chan_no = 0;

     client->pdev  = NULL;

     /*

      * Initialize the callback pointer

      * A value of NULL here would indicate polling is the

      * preferred method for querying transactions.

      */

     client->edma_callback = test_client_event_callback;

     /* Register */

     if(dma_funcs.edma_client_register(client) == 0)

     {

         /* Allocate a channel */


          chan = dma_funcs.edma_chan_alloc(kern_client);


          if(chan != NULL)
```

```
                {
                  dma_funcs.edma_chan_release_(chan)
                 }
            }
        }
```

3.3.6 edma_chan_release Function

The **edma_chan_release** function releases a DMA channel resource when a client no longer wishes to use it.

```
void edma_chan_release(struct edma_chan * chan)
```

Table 8. edma_chan_release Parameters

Parameter	Description
*chan	DMA channel resource handle.

3.4 Memory Transactions

Client drivers will supply the **edma_memcpy** function with a value constructed by logically “OR-ing” defined constants that represent particular transfer modalities for the source and destination addresses. The following list contains the constants representing the memory transaction modalities present in the EDMA_W platform.

Table 9. Memory Transaction Modalities (Sheet 1 of 2)

Constant	Description
SRC_DWORD_ALIGNED	Source address is DWORD aligned.
SRC_CACHE_ALIGNED	Source address is aligned on a cache line boundary.
DST_DWORD_ALIGNED	Destination address is DWORD aligned.
DST_CACHE_ALIGNED	Destination address is aligned on a cache boundary.
SRC_NONCOHERENT	For the source address, no FSB snoop cycle will be issued on behalf of EDMA memory accesses.
SRC_COHERENT	For the source address, a FSB snoop cycle can be issued on behalf of EDMA memory accesses.
DST_NONCOHERENT	For the destination address, no FSB snoop cycle will be issued on behalf of EDMA memory accesses.
DST_COHERENT	For the destination address, a FSB snoop cycle can be issued on behalf of EDMA memory accesses.
SRC_MEM	Source address is in local memory.
DST_MEM	Destination address is in local memory.
DST_IO	Destination address is in memory mapped I/O.
SRC_INC	Source address is incremented as data is read.
SRC_DEC	Source address is decremented as data is read.
SRC_BUFFER_INIT	Source address contains a constant value to be written to destination address.

Table 9. Memory Transaction Modalities (Sheet 2 of 2)

Constant	Description
DST_INC	Destination address is incremented as data is written.
DST_CONST	(EDMA FIFO mode) 1, 2, or 4 bytes are sent repeatedly until the size count is satisfied.
GRAN_1_BYTE	1 byte granularity of the EDMA FIFO mode described above.
GRAN_2_BYTE	2 byte granularity of the EDMA FIFO mode described above.
GRAN_4_BYTE	4 byte granularity of the EDMA FIFO mode described above.
TC_0 – TC_7	Traffic classes.

3.4.1 edma_memcpy Function

The **edma_memcpy** function is used to initiate a DMA memory copy operation on a DMA channel. A “cookie”, a value associated with the particular transaction, is returned. An interrupt will be generated upon reaching the end of chain (series of transfers), and the client will receive a callback via the edma_callback_t value if the client supplied a non-null callback value in the client struct. If a hardware error occurs during processing, the client will receive a callback. The client can poll that the operation is complete by using the **edma_is_complete** call. The dma_addr_t type is a Linux kernel type which holds valid DMA addresses for the platform.

```
edma_cookie_t edma_memcpy(

    u32 memory_ops,

    struct edma_chan *chan,

    dma_addr_t dest, /* edma_addr_t from #include <linux/pci.h */

    dma_addr_t src,

    size_t size,

    unsigned long user_data

);
```

Table 10. edma_memcpy Parameters

Parameter	Description
memory_ops	Contains a value that is the result of the client logically “OR-ing” defined constants that represent potential transfer modes that the EDMA driver will use to program the EDMACTRL register. The EDMA driver does not check this value to determine if it is one of the valid transfer mode combinations for the source and destination addresses. Misuse of this value may cause a serious error.
*chan	DMA channel resource handle.
dest	Destination address, an address in physical memory.
src	Source address, an address in physical memory.
size	Size of the memory copy operation in bytes, up to 16 MBytes per block for the EDMA_W platform.
user-data	This client-defined value is returned as part of the edma_callback. This integer type will always be large enough to store a pointer value, but may not be large enough to store a physical address.

Return Values:

- An integer value of greater than 0 indicates a valid cookie value.
- A memory size that requires more than the 16 descriptors or the number of free descriptors available will return an invalid cookie value (ERROR_OUT_OF_DESCRIPTORS). See the Section 3.4.2, “Interrupts and Descriptor Chains” on page 20 for more information.

Example Usage:

```
void test_client_event_callback(struct edma_client * client, struct edma_chan *
chan, edma_cookie_t cookie, enum edma_status_t status, unsigned long user_data);


struct edma_client * client = NULL;

struct edma_client * chan = NULL;

unsigned int memory_ops = SRC_DWORD_ALIGNED | DST_DWORD_ALIGNED | DST_MEMTOIO
|SRC_MEMTOMEM | DST_COHERENT | SRC_COHERENT;

void * src = kmalloc((sizeof(char) * 0x0400), SLAB_HWCACHE_ALIGN);

void * dest =  kmalloc((sizeof(char) * 0x0400), SLAB_HWCACHE_ALIGN);

dma_addr_t phys_src = (dma_addr_t)virt_to_phys(src);

dma_addr_t phys_dst = (dma_addr_t)virt_to_phys(dest);

size_t transfer_size = 0x0400;

client = dma_funcs.edma_client_alloc();

/ * report error */

if(client == NULL)

{

    printk(KERN_INFO "Problem allocating client\n");

}

else

{

     /* Channel that the Client will be requesting. */

     client->chan_no = 0;

     client->pdev  = NULL;

     /*

      * Initialize the callback pointer

      * A value of NULL here would indicate polling is the

      * preferred method for querying transactions.

      */

     client->edma_callback = test_client_event_callback;
```

```
          /* Register */

          if(dma_funcs.edma_client_register(client) > -1)

          {

               chan = dma_funcs.edma_chan_alloc(kern_client);


               if(chan != NULL)

                {

                cookie = dma_funcs.edma_memcpy(memory_ops, chan,

                phys_dst, phys_src, transfer_size, 0);

                }

     }
```

3.4.2 Interrupts and Descriptor Chains

The EDMA driver uses a linked list of sixteen descriptors (data structures which store source and destination memory locations) to facilitate memory transactions. The descriptors comprise a descriptor chain which is created by the EDMA driver upon loading and initialization. The number of descriptors used in a particular memory transaction is dependent on the size of the transaction. The EDMA_W platform may handle a maximum memory size of 16 MBytes per descriptor; hence, a 32 MByte transaction would be split over two descriptors.

Each descriptor has a “cookie value”, an integer value greater than or equal to zero, which identifies a particular memory transaction and aids in tracking the transaction’s progress. The cookie value is returned to the client upon a call to edma_memcpy and may also be used by a client to poll the driver about status of memory transactions.

Because there are a finite number of descriptors in the descriptor list, the driver maintains a list of used and free descriptors. Used descriptors are reclaimed after a memory transaction has completed. Information about a particular transaction may expire, as the descriptors are reused.

The EDMA driver is configured to handle an interrupt only when an end of chain or abort condition occurs. An end of chain condition occurs when the driver has finished processing the last assigned descriptor in the descriptor list. When an end of chain event occurs, the EDMA driver’s interrupt handler will execute callbacks to the client driver for each completed transaction if a callback function pointer was supplied in the client struct. Alternatively, if no callback pointer was made available, the client driver may use the polling function to determine if a particular transaction completed successfully.

An abort condition occurs if the driver experiences serious errors during a memory transaction. As with the end of chain condition, the EDMA driver’s interrupt handler will execute a callback to the client driver if a callback function pointer was supplied in the client struct. In addition, if there is a transaction in the descriptor list after the problematic transaction, the EDMA driver will restart the DMA engine to execute the next transaction.

Single memory transactions that require more than 16 descriptors will produce an error message with the edma_memcpy routine. These extremely large transactions should be broken into multiple transactions. In addition, if the EDMA driver does not have enough free descriptors to service a particular memory transaction, an error will be produced.

3.5 Polling and Callbacks

The following sections present details about polling and callback mechanisms discussed in the previous interrupt sections.

3.5.1 edma_callback_t Typedef

The **edma_callback_t** typedef is used to define the DMA callback, a pointer to a function, in the client structure. A DMA callback function is used to notify the client driver of an error or the completion of a transaction. Callback functions should be designed to execute and return quickly. See the section edma_client for more details.

```
typedef void (* edma_callback_t)(

    struct edma_client *client,

    struct edma_chan *chan,

    edma_cookie_t cookie,

    edma_status_t status,

    unsigned long user_data

);
```

Table 11. edma_callback_t Parameters

Parameter	Description
*client	Pointer to a client instance structure.
*chan	DMA channel resource handle.
cookie	DMA operation identifier.
status	Status of the DMA operation.
user_data	The client defined value passed to edma_memcpy is returned here.

Example Usage:

```
void test_client_event_callback(struct edma_client * client, struct edma_chan *
chan, edma_cookie_t cookie, enum edma_status_t status, unsigned long user_data)

{

    /*

        Include in this function checks for error

        status…

      */

        if(status == EDMA_ERROR)

        {

       /* Call to an error handling function which calls

               edma_get_errors.
```

```
            */
        }
}
client->edma_callback = test_client_event_callback;
```

3.5.2 edma_is_complete Function

This function is used to poll the status of a DMA memory copy operation. Because errors always result in a dma_callback if a callback function was assigned and DMA operations will always complete in order, it is valid for a client to use a single call to this function to determine if multiple DMA operations have completed rather than check the status of each operation individually.

```
edma_status_t edma_is_complete(
    struct edma_chan *chan,
    edma_cookie_t cookie
);
```

Table 12. edma_is_complete Parameters

Parameter	Description
*chan	DMA channel resource handle.
cookie	DMA operation identifier.

Return Values:

EDMA_ERROR = -1	Error occurred in transaction.
EDMA_SUCCESS=0	Transaction completed successfully.
EDMA_IN_PROGRESS=1	Transaction in progress.
EDMA_NO_RECORD=2	No record for transaction is available.

Example Usage:

```
do
{
     status = dma_funcs.edma_is_complete(chan, cookie);
     printk("ClientDriver: status %x \n", status);
}while(status == EDMA_IN_PROGRESS);
```

3.5.3 edma_get_errors Function

EDMA errors are considered non-fatal to the system because they cause the EDMA engine to stop the process. When an error occurs the EDMA_FERR is set and locked; after it has been locked, all subsequent errors are placed in the EDMA_NERR register. If a DMA operation completes with status EDMA_ERROR, this function is used to retrieve the contents of the EDMA_FERR register. A call to this function clears the EDMA_FERR.

```
u32 edma_get_errors(
    struct edma_chan *chan,

    edma_cookie_t cookie

);
```

Table 13. edma_get_errors Parameters

Parameter	Description
*chan	DMA channel resource handle.
cookie	DMA operation identifier.

Return Values:

- The lower 16 bits contain the value of the EDMA_FERR register for the DMA channel and the upper 16 bits contain the descriptor in which the problem occurred.

Example Usage:

```
do

{

     status = dma_funcs.edma_is_complete(chan, cookie);

     printk("ClientDriver: status %x \n", status);

     if(status == EDMA_ERROR)

        {

         err_val = dma_funcs.edma_get_errors(chan, cookie);

        }

}while(status == EDMA_IN_PROGRESS);
```

§§

4.0 Modifying the Driver

The following sections describe several potential modifications of the EDMA driver behavior defined in the previous sections.

4.1 Loadable Modules

The EDMA driver is registered as a character driver. As written, the driver uses the register_chrdev function in the edma_init function to assign a major number to the driver and assign the file_operations structure, which is defined at the top of the dma.c file.

```
int register_chrdev(unsigned int major, const char * name, struct file_operations *
fops)
```

After the driver is loaded with the insmod program, the major number of the driver can be seen in the /dev directory with the ls –l command. A list of major numbers and corresponding devices can be found in the file documentation/devices.txt. The major number for this driver was chosen at random, but this is not a very good permanent solution.

Dynamic allocation can be used to obtain a major number for the device. Scripts that accomplish dynamic allocation are available on the web or in the texts listed in the Appendix. Using dynamic allocation will adversely affect the ability to use the load-on command feature.

4.2 Client/Channel Relationships

A single client can allocate multiple channels, but a channel is allocated to a single client at any given time. As described in previous sections, clients request a channel by filling in the channel number of the client structure. The edma_chan_alloc function uses this number to check if the particular channel is in use. This behavior can be altered by removing the channel number data member from the structure and simply tracking channel usage in the channel allocation function.

4.3 Descriptor List

As written, the EDMA driver allocates 16 descriptors per channel and maintains these lists as a global variable in the dma.c file.

```
static struct edma_descriptor_list g_descriptor_lists[MAX_NO_CHANS];
```

The descriptor type edma_descriptor_list can be found in the dma_internal.h file. Pools of memory for the descriptor lists are allocated with kmalloc in create_internal_structures. In order to modify the number of descriptors in the EDMA driver, alter the DESCRIPTOR_NO constant in dma.h.

§§

Appendix A References

- *Linux Device Drivers*, *2nd Edition* by Alessandro Rubini and Johnathon Corbet.

§§