Multichannel DMA API User's Guide

1 Introduction

The multichannel API provides an API for using the multichannel DMA on the MPC8220, MCF547x, and MCF548x.

The multichannel DMA is a centralized on-chip DMA controller which can control data transfers with minimal intervention from the processor core. It can transfer data to and from various sources and destinations such as DRAM, on-chip SRAM, and on-chip peripherals.

The DMA processes a series of instructions called "descriptors" to determine its execution much like a CPU processes instructions. A series of descriptors that is designed to perform a specific operation is called a "task" and is much like a function in a programming language. Variable parameters can be passed to each task by using a fixed size variable table which is assigned to each channel of the DMA. The DMA can also perform operations on data while it is transferring it. These operations are determined in part by the task descriptors and also by another set of descriptors called function descriptors, which are referenced by the task descriptors and stored in another fixed size table called a "function

Table of Contents

descriptor table." Lastly, the DMA only executes the task of one particular channel at a time, so it will often perform a task swap in much the same way that an operating system might swap out processes when one process goes to sleep or a higher priority process needs to execute. When this occurs, the DMA saves relevant information which it may need to restart the current task, in a context save table.

The purpose of the multichannel DMA API is to provide a small set of precoded tasks combined with some C functions and structures, which allow the user to easily use these tasks and manage the tasks and associated tables.

1.1 Features

The multichannel DMA API provides simple API-level calls for typical DMA functions, as well as for some more exotic DMA actions. Examples of these DMA capabilities include the following:

- Single-buffer DMAs, wherein the source and destination addresses are specified in the call parameters
- Multi-buffer DMAs, wherein the addresses are specified in buffer descriptors stored in memory
- DMAs in byte (8-bit), word (16-bit), or longword (32-bit) transfer sizes
- DMAs initiated by any Comm subsystem device
- The ability to jump to a new buffer upon detecting a packet/frame/cell/datagram/etc. boundary in received data
- Injection of packet, frame, cell, datagram, etc. boundaries into transmitted data
- Versatile source- and destination-pointer increments, including negative increments and non-contiguous increments (for example, increments of magnitude larger than the transfer size), and different source and destination increments
- The ability to query progress of some DMAs while they execute
- The ability to pause, resume, or kill a DMA
- Interrupt generation on any buffer boundary, independent of framing

Compile time extensions to the API allow for the usage of the following features in some cases:

- CRCs of data as it is moved
- Byte- and/or bit-reordering of data as it is moved

The multichannel DMA API does not handle the following:

- Peripheral initialization
- SRAM memory management
- Interrupt handling
- Cache management
- Virtual address translation

1.2 Document Conventions

In C code fragments, variable size and sign are indicated by the following typedefs:

- u32 is a typedef for an unsigned 32-bit quantity
- s8 is a typedef for a signed 8-bit quantity
- s16 is a signed 16-bit quantity
- int is exactly what it means in C, and implies that the size is left up to the compiler.

For the purpose of this document, 32-bit quantities are referred to as "longwords," 16-bit quantities as "words," and 8-bit quantities as “bytes.”

2 API Usage

The API provides a set of functions for such operations as task image and DMA initialization, task start, transfer progress query, and so on. These functions as well as the pre-coded task images (also known as task descriptor tables), are compiled and linked into applications, device drivers, or operating system code. These task images and related functions take several forms depending on the purpose of the task, such as which peripheral it was intended to operate with.

2.1 Building the API

The multichannel DMA API is provided as C source code and header files. To use the multichannel DMA API functions do the following:

- #include <MCD_dma.h> in the sourcecode file
- Add the appropriate DMA calls to your C code

2.1.1 Files

The following files are provided:

- MCD_dma.h—contains information necessary for the user and the API. Definitions of structures and flags and prototypes for the user- callable functions
- MCD_dmaApi.c—implements the bulk of the API code such as initialization, task determination and start, status, progress query and so on
- MCD_tasks.c—contains the DMA code - task descriptor tables, function descriptor tables, and space allocated for variable tables and context save spaces
- MCD_tasksInit.c—individualized code for setting up the variable tables of each different pre-coded task
- MCD_tasksInit.h—prototypes for the functions in MCD_tasksInit
- MCD_progCheck.h—defines constants used for progress checking

The following files should be compiled and linked:

- MCD_dmaApi.c
- MCD_tasks.c
- MCD_tasksInit.c

2.1.2 Compile Time Options

The API currently includes one compile time option.

2.1.2.1 MCD_INCLUDE_EU

Defining MCD_INCLUDE_EU will include task code, task structures, and C code which will allow the user to use the funcDesc parameter of MCD_startDma. This allows the user to perform operations on data as it is being transferred. This will increase the overall space required for the DMA tasks and tables. The user must balance the effect of this additional size with the benefits of the additional feature set.

2.2 Initialization

The user must initialize the DMA API by calling the function MCD_initDma. The primary purpose of this function is to set up internal variables which tell the API where the multichannel DMA register set is located, and where the task descriptor tables and other associated tables are located in memory. This function can be directed to move the set of tables to a location in memory (the preferred location being the on-chip system SRAM) and to set up some global behaviors which can affect all tasks run by the DMA.

2.3 Types of DMA Tasks

Starting a DMA task is accomplished by calling the function MCD_startDma. Because there are various tasks, the type of task that is run depends on the value of the flags and funcDesc parameters passed to the function call.

2.3.1 Ethernet Tasks

The API contains specialized tasks to transfer data to and from the Ethernet receive and transmit FIFOs. These tasks use a set of buffer descriptors arranged in a ring. The buffer descriptors are described in Section 2.5.2, “Format for Ethernet DMAs.” The buffer descriptor ring must be initialized before calling MCD_startDma with either the MCD_FECTX_DMA or MCD_FECRX_DMA flags defined. The funcDesc parameter has no effect on these tasks. These tasks exit when encountering an unready buffer descriptor and can be restarted at that buffer descriptor by using the function call MCD_contInDma.

2.3.2 Single Buffer Tasks

The DMA API allows for the use of single buffer and multi-buffer DMAs. For single buffer DMAs, the nature of the transfer is decided entirely by the parameters passed in the MCD_startDma call. A single DMA task is run when the flags parameter has MCD_SINGLE_DMA defined and when the Ethernet flags are not defined.

2.3.3 Multiple Buffer (Chained) Tasks

For multiple buffer, or chained, DMAs, the user provides a set of buffer descriptors that are arranged in a chain or linked list. The buffer descriptors are described in Section 2.5.1, “Format for Chained DMAs”.

The MCD_startDma parameters are again used to determine which microcode task will be run and to set up the channel, but the underlying microcode task will evaluate each buffer descriptor's flags to determine the behavior for each individual buffer descriptor. A chained DMA task is run if neither MCD_SINGLE_DMA is defined nor the Ethernet flags.

2.3.4 Execution Units

The multichannel DMA currently supports an execution unit that can perform manipulations on data while it is being transferred. This execution unit is known as the LURC (Logic Unit with Redundancy check). LURC support for data manipulation during transfer is provided when the compile time option MCD_INCLUDE_EU is defined. There are additional special considerations for this operation noted throughout the document. The execution unit tasks are analogues of the single- and multiple- buffer tasks and are selected when the following are true:

- MCD_FECTX_DMA and MCD_FECRX_DMA are not defined in the flags parameter.
- The funcDesc parameter indicates that a byte swap, bit swap, or CRC should be performed.

As shown above, the Ethernet tasks do not support data manipulation during transfer. Also note that all buffers in a multiple buffer DMA will perform the same operation on data which is transferred. The operation is not selectable on a per-buffer basis.

2.3.5 Framing

Some peripherals in the communications subsystem, such as Ethernet and USB, support framed or packetized modes of operation. This type of data encapsulation can also be referred to as "cell" or "datagram." For the purposes of this document, the term "frame" will be used. To support this functionality, the DMA and peripherals use internal signals to indicate where the end of frames occur, both when transmitting data from DMA to peripheral and when receiving data from peripheral to DMA.

In the case of peripheral-to-memory DMAs, it is important to understand that the DMA is under the control of the activity on the line. That is true not only in the sense that DMA transfers occur in response to data reception, but also in that DMAed data also follows the grouping of frames in the data on the line. If, while receiving a buffer of data, a frame boundary is detected on the incoming data stream, the DMA is terminated at that boundary. The actual length of the frame received can be retrieved by calling MCD_XferProgrQuery() after the DMA is complete or by using the lastDestAddr which is written by the DMA in chained operation.

If the protocol the peripheral receives is like a UART or IEEE 1284 interface in that it does not involve frames, then the above-described framing considerations do not apply; the DMA transfers the requested amount of data.

For memory-to-peripheral DMAs, it is up to the user to indicate to the DMA whether framing will be used. The DMA can then indicate to the peripheral that data is framed and on what boundary.

Also, this does not apply to memory-to-memory or memory-to-non-framing-peripheral DMAs since, under those circumstances, the application running on the CPU defines the grouping of the data.

2.3.6 Specialized DMAs

As shown above, not every option available applies to every DMA task. Some DMAs do not conform to the more general notions presented in this document since they are optimized either for performance or to work with a special peripheral. Many of the options that are available for the default DMA call may not be available for specialized DMAs, which handle devices such as the Fast Ethernet Controller (FEC).

2.4 System Considerations

There are several system considerations which the user must be aware of in order to better use the API.

2.4.1 Interrupt Handling

This API does not address how to handle DMA interrupts; it only defines how to generate them. There is one interrupt line output by the multichannel DMA which is an input to the on-chip interrupt controller. The user can control the masking of interrupts from the different channels of the DMA in the multichannel DMA interrupt mask register. Additionally, the user must read the multichannel DMA interrupt pending register to determine which DMA channel is causing the interrupt. See the appropriate hardware reference manual for more information.

2.4.2 Task Prioritization

The API allows user software to set the priority of each DMA task every time a new DMA is begun on a particular channel with the call of MCD_startDma. Since the multichannel DMA does not provide a fairness algorithm in its arbiter, the user must set the priorities carefully with the knowledge that higher priority tasks may block lower priority tasks indefinitely as long as their initiators are asserted. This API uses the DMA's Task Priority mode instead of Initiator Priority mode. In addition, enabling of tasks takes priority over running of tasks in order to avoid locking out of a high priority task because another task is currently running.

2.4.3 Virtual Memory and Physical Addresses

The DMA controller works entirely in terms of physical addresses, so logical or virtual addresses need to be translated to physical addresses, and any discontiguousness in the physical address space must be accounted for. In any paged memory system, sequential pages in logical address space could be mapped to non-sequential addresses in physical address space. The user must account for this when calling API functions and when programming buffer descriptors.

2.4.4 Task Code and Buffer Descriptor Memory

The MultiChannel DMA requires the user to provide memory for storage of the DMA task code (task descriptor tables) and other supporting structures (task table, variable tables, function descriptor tables, and context save spaces). These things are intended to be stored in the on-chip system SRAM for better performance. In addition, the API itself requires the user to provide memory for buffer descriptors when used. These can also be stored in the internal SRAM. For some implementations this may be required due

to optimizations in the interface between the multichannel DMA and the external memory buses. The task code and buffer descriptors should be placed in non-cacheable memory. The API does not handle memory management for these areas of memory.

2.4.5 Memory-to-Memory DMAs

Memory-to-memory DMAs are unique in that, unlike peripherals, they are not governed by any specific initiator. The Always initiator can be used, but should be used carefully. Since the Always initiator is always asserted, any tasks which use it to govern the transfer of data can block lower priority tasks throughout the duration of the task. Another option is to use the comm timers as initiators for memory to memory DMAs. The comm timers allow the software to vary the periodicy of the initiator. This would allow lower priority tasks to operate when the initiator is deasserted.

2.5 Buffer-Descriptors

The API supports two different buffer descriptor formats: one multiple buffer chained DMAs and one for use with ethernet.

2.5.1 Format for Chained DMAs

The buffer descriptors for chained DMAs are of the following format:

```
typedef struct MCD_bufDesc_struct MCD_bufDesc;

struct MCD_bufDesc_struct

{
        u32 flags;         /* flags describing the DMA (explained below) */
        u32 csumResult;    /* checksum resulting from checksumming performed
                              since last checksum */
        s8  *srcAddr;      /* the address to move data from */
        s8  *destAddr;     /* the address to move data to */
        s8  *lastDestAddr, /* the last address written to (see note below) */
        u32 dmaSize;       /* the number of bytes to transfer independent of
                              the transfer size reset */
        MCD_bufDesc *next; /* next buffer descriptor in chain */
        u32 info;          /* private information about this descriptor; DMA
                              does not affect it */
};
```

The values of the flags member of the buffer-descriptor structure are similar to those described for the flags parameter of the MCD_startDma call:

- **MCD_INTERRUPT** to cause the DMA channel to generate an interrupt after completion of this buffer.
- **MCD_BUF_READY** to indicate that this buffer has been setup by the CPU code, and is now under the DMA's control. When this buffer is DMAed this indication is reset (bufferDesc.flags & MCD_BUF_READY == 0). Upon encountering a buffer descriptor that is not ready for the DMA, the DMA function completes execution. When execution completes, the MCD_XferProgrQuery() function returns a pointer to this non-ready buffer descriptor.
- **MCD_END_FRAME** to instruct the DMA to terminate the frame on the transmitted data stream at the end of this buffer. If the DMA is not transmitting data, or if the peripheral involved doesn't deal with frames, then this has no effect.
- **MCD_CRC_RESTART** to empty out the accumulated checksum prior to performing the DMA. In chained DMAs the CRC accumulated since the CRC was last reset is placed into the csumResult field of the buffer descriptor containing the MCD_CRC_RESTART indication. This flag is only used by tasks that are provided when MCD_INCLUDE_EU is defined.

Any other flags which apply to MCD_startDma are not applicable to the buffer descriptors. They must not be used with the buffer descriptors or unpredictable behavior may occur.

It's important to note that a NULL pointer to the next buffer is not sufficient to stop the DMA. This is a common technique for determining the last element of a linked list in software. However, only the lack of MCD_BUF_READY indication in the flags of the buffer descriptor will stop the DMA. This has the disadvantage that, in the case of a terminated linked list, a dummy end-indicator buffer descriptor is needed. This end-indicator buffer need not be entirely a dummy though. If the flags indicate that the CRC is to be restarted, then its csumResult structure member is where the resulting CRC is written.

As alluded to earlier, the MCD_END_FRAME indication is valuable on a buffer-by-buffer basis so that frames of data can be constructed from more than one buffer. For example, this allows there to be one single contiguous array of raw data in memory, which can be logically partitioned into (hypothetically) 1K pieces. One or more very short separate buffers can then be provided to hold protocol headers for each 1K piece of the contiguous memory image, and the resulting packets gathered together into the correct frame format.

The end-frame indication would only be set on the last buffer that comprises the frame. This ability reduces the amount of time spent on data copies because the raw data is kept in contiguous form.

The MCD_INTERRUPT interrupt-or-not indication can be used to achieve several effects:

- Interrupt after every buffer, by setting that indication in all of the buffer descriptors.
- Interrupt after the combination of buffers that form a single frame, but avoid interrupts on the individual components of a frame (e.g., streams headers).
- Interrupt only after all DMAs in a chain are completed.

Similarly, the MCD_CRC_RESTART indication can be used to CRC only relevant portions of a frame/packet of data, by restarting the CRC on the buffer associated with the data of interest. This flag is only valid if the API is compiled with MCD_INCLUDE_EU defined.

The member lastDestAddr denotes the first address in the destination buffer not DMAed to as the buffer completed (or it is the same as destAddr if the destination increment is 0). More simply stated, it is the post-increment final address pointer value used in this buffer's DMAing.

Conceptually, lastDestAddr would be given by the formula destAddr + destIncr * (dmaSize / xferSize), were it not for one situation. In the case of peripheral-to-memory DMAs, the DMA is under the control of the activity on the line. If, while receiving a buffer of data, a frame boundary is detected on the incoming data stream, DMA to that buffer is terminated at that boundary in order to correctly reflect the end of that frame in the end of that buffer. The next frame begins in the next buffer. However, this does not work the same if any manipulations of the data are being performed during the transfer. If byte or bit reordering, or CRCing is being done on the data, lastDestAddr will always be on the next xferSize boundary. The MultiChannel DMA always performs these manipulations on xferSize chunks of data, and in fact strictly on longword chunks in the case of byte reordering. Once the data manipulation hardware is engaged, it has no means of knowing where within an xferSize chunk a frame boundary occurs. However, if the DMA is done with no data manipulation (i.e., funcDesc parameter == MCD_NO_BYTE_SWAP | MCD_NO_BIT_REV | MCD_NO_CSUM or MCD_NO_BYTE_SWAP | MCD_NO_CSUM), the frame boundaries are always detected down to byte resolution. Since frame boundaries are detected at xferSize resolution when data is being manipulated, the frame boundaries will be detected to byte resolution when xferSize is byte (i.e., 1), whether the DMA manipulates the data it moves or not. If frames are always guaranteed to have lengths in multiples of 4 bytes, then an xferSize of 4 with data manipulation will work as expected. In the case of single-buffer, non-chained DMAs, this frame-size information can be retrieved by calling MCD_XferProgrQuery() after the DMA is complete.

If the protocol the peripheral receives is like a UART or IEEE 1284 interface in that it does not involve frames, packets, cells, or the like, then the above-described framing considerations do not apply; the DMA transfers exactly the amount of data requested.

Also, this does not apply to memory-to-memory or memory-to-peripheral DMAs, since, under those circumstances, the application running on the CPU defines the grouping of the data rather than the being defined by the data format on the line.

Chaining buffer descriptors in a linked list addresses the following DMA needs:

- Allowing the CPU to insert data buffers into the chain easily, such as to building up a sequence of streams headers, or just simply adding additional data movements to the DMA's "to do list," so to speak, while the chained DMA is in progress.
- Allowing circular linking to allow one or more buffers to provide a double-buffering sort of scheme. For example, the DMA can interrupt on every buffer as with an unchained DMA, but the DMA can immediately work on the next buffer in parallel with the CPU processing the one just completed.

This double-buffering approach can be extended to a flexible buffer-queue scheme.

2.5.2 Format for Ethernet DMAs

The FEC tasks implemented in the API use a buffer descriptor format similar to that implemented for Ethernet controllers on other Freescale devices such as the PowerQUICC family and some ColdFire

processors. This buffer descriptor format differs significantly from the chained multichannel DMA buffer descriptor format.

```
typedef struct MCD_FEC_bufDesc

{

      u16 statCtrl;

      u16 length;

      u32 dataPointer;

}MCD_FEC_BD;
```

There are several major differences when using this buffer descriptor format and the associated tasks:

1. Implemented as a buffer descriptor array, or ring, instead of a chain. There is no pointer to the next buffer descriptor; it is assumed to be right after the current buffer descriptor. The ring is instructed to wrap back to the beginning of the ring when the buffer descriptor has the MCD_FEC_WRAP bit set.
2. There is no support for data manipulation while transferring data. The FEC module itself computes CRCs for each frame.
3. There is no support for progress reporting.

The statCtrl field is similar to the flags field for the multichannel DMA buffer descriptor. The major difference is that statCtrl is only 16 bits wide whereas the flags for the chained DMA are 32 bits.

The buffer descriptor fields are described as follows:

- **MCD_FEC_BUF_READY** indicates that the buffer is owned by the DMA. The user should set this bit in the buffer descriptor when data is ready for transmit or if the buffer is ready to receive data from the Rx channel.
- **MCD_FEC_WRAP** indicates that the current buffer descriptor is the last buffer in the ring. The DMA will begin processing buffer descriptors from the beginning of the ring after it finishes with the BD with the wrap bit set.
- **MCD_FEC_END_FRAME** indicates that the DMA should mark the last piece of data transferred in this data buffer as the end of a frame. This flag has no effect on the Ethernet receive task.
- **MCD_FEC_INTERRUPT** indicates that the DMA should generate an interrupt which indicates that the DMA has finished transfer the Tx data to the FEC Tx FIFO or writing of Rx data to the destination buffer. For transmit, this does not indicate that the data has completed transferring on the external Ethernet, but only that the frame has completed being transferred to the FEC Transmit FIFO.

The statCtrl field is manipulated by both transmit and receive tasks when data transfer for the buffer is completed. For transmit, the ready bit is reset to zero to indicate that the buffer is no longer ready. The other bits remain set. For receive, the DMA will update the statCtrl field and the length field with the contents of the control/status word which is read out of the FEC receive FIFO. The status bits in the statCtrl

field are defined by the definition of that word as defined in the hardware manual. The state of the wrap and interrupt bits are preserved.

For transmit, the length field is the amount of data to transmit. The dataPointer field is a pointer to a data buffer. This dataPointer must be a 32-bit aligned address. If the user's data buffer is not aligned on a 32-bit boundary, it needs to be copied to one that is.

For receive, the length field is the maximum frame size which the DMA should allow for that buffer. It should be programmed with a value of 2,048 bytes, which is the maximum size the FEC will allow before truncation. As noted, the length will be overwritten with data from the FEC which states how much data was actually received for that buffer. The receive task does not support fragmenting a received frame across more than one buffer; each buffer must be as large as the largest possible frame (2 Kbytes).

The dataPointer field is a pointer to a data buffer. This dataPointer must be a 32-bit aligned address.

2.6 DMA Direction

The API provides limited support for DMA transfers occurring in both positive and negative address increments. This is controlled by the MCD_startDma call parameters srcIncr and destIncr, and the direction is determined by the sign of the parameter. Whether transferring forward or backward, the address provided to the DMA should be xferSize aligned to the first xferSize entity which will be transferred. For example, if the user wanted to transfer the memory range 0x1002 through and including 0x101f in 16-bit chunks forward from 0x1002, then the starting address would be 0x1002. However, if the user wanted to transfer the same memory range backwards starting at address 0x101f in 16-bit chunks, then the starting address would be 0x101e. The DMA will by default do xferSize accesses, and the first address of a 16-bit transfer is 0x101e, with the bytes at addresses 0x101e, and 0x101f being transferred first.

2.7 Progress Query

The API provides two methods for monitoring how much data has been transferred by the single-buffer and multiple-buffer tasks: using the lastDestAddr field of the chained buffer descriptor and using the MCD_XferProgrQuery function. This functionality is implemented by saving to memory the values of DMA internal registers to which the user does not have access. There are several limitations to the monitoring provided by this API which are imposed by the hardware and for efficiency reasons.

One limitation is that progress reporting only indicates the progress of the DMA. It may not accurately reflect the actual progress made by the entire system due to various system buffers and FIFOs. For example, a receiver peripheral may have data stored in its receive FIFO which the DMA has not read yet. Additionally, the progress reporting cannot account for the fact that receive data may be stored in a line buffer on its way to DRAM or other memory storage. For this reason, progress query results may either be a little behind when reflecting how much data has been received by the system or a little ahead of what data has actually when transferred to memory. Similarly, progress results for transmit DMAs may be a little behind how much data has been read from memory but a little ahead of the data flow on the external bus.

Another limitation is that the progress query information available to the user is not continuously updated. Saving progress information after every single data beat is transferred would severely degrade performance. Therefore, the progress information is only updated at specific points during DMA operation. Progress-query information is updated at the following points:

- Whenever the initiator of the task negates
- When some other channel's activity pre-empts it
- When a DMA is paused or killed (lastDestAddr of buffer descriptor will not be updated when a task is killed)
- Upon completing each buffer

Users must therefore be aware that DMAing at high priority using an initiator that is always asserted will move that data very quickly, but the DMA progress will rarely or never be updated until the DMA is done.

Users should also bear in mind that saving progress-reporting information to memory itself takes a small number of tens of clocks, depending upon the speed of the memory into which it is being saved. However, only four consecutive memory writes of the information the hardware saves during these few tens of clocks is used by MCD_XferProgrQuery(). If a DMA's initiator negates, thereby causing the DMA to update progress, and if the DMA's initiator reasserts very shortly thereafter while it is still saving this information, it will not waste time saving the rest of that information.

2.7.1 Partial Updates

If a task's initiator deasserts and reasserts very quickly, there is the possibility that the information used for MCD_XferProgrQuery may be partially updated. The most common result of aborting this update is that the four memory writes used by MCD_XferProgrQuery() do not happen (i.e., it does not get that far), because the DMA is, effectively, continuously running. The progress-query information is therefore not updated.

However, a much rarer, but not impossible, result of DMA initiators changing very quickly is corruption of the progress-query information. This theoretically can occur if, when the initiator reasserts, the save of this information aborts between the first and last of the four writes of the progress information. MCD_XferProgrQuery() makes sure that it does not read progress information simultaneously while the DMA is writing it, but the API has no means to detect when it has been partially written.

MCD_XferProgrQuery() does, however, have logic to reduce the likelihood of providing erroneous information if the progress-reporting information should become self-inconsistent. For example, the last of the four pieces of information saved (the most likely to get cut off) can always be calculated from the other three, and therefore always is.

The risk of the progress-reporting information being partially saved cannot occur in a change from servicing one channel to servicing another channel. This is only at risk of happening when transfers for one single channel cease for a very short period of time and then resume.

The solution to this concern is to make certain that DMA initiators are not allowed to negate and reassert faster than several tens of clocks. These sorts of system-tuning-parameter settings are almost always highly desirable, normal practice to keep from pointlessly thrashing the DMAs between channels every few tens of clocks. In the case of I/O devices, this means making sure that the FIFOs' high- and low-water marks are set so that the I/O device takes at least a few tens of clocks to fill or empty. For memory-to-memory DMAs, which are generally initiated by timers, setting the timer-low time, that is, the timer period * (1 - pulse width), is greater than a few tens of clocks.

2.7.2 Buffer Descriptor Update

The value of the lastDestAddr field returned by MCD_XferProgrQuery is calculated in the same way as the lastDestAddr field of the chained buffer descriptor; they both reflect the state of an internal register which keeps track of progress. See Section 2.5.1, "Format for Chained DMAs" for discussion of how this value is calculated. Therefore it behaves the same way in respect to its calculation when dealing with framing peripherals. However, it is important to note that the lastDestAddr field of the buffer descriptor may be updated less often than the same value returned by MCD_XferProgrQuery. They use the same internal DMA register, but the lastDestAddr field of the buffer descriptor must be written to manually by the task whereas the value used by the progress query function is automatically saved by the hardware at certain points. For that reason, if the task is paused or killed, the lastDestAddr of the buffer descriptor may not be accurate.

2.8 Register Control

The API controls multichannel DMA registers and data structures in memory associated with starting a DMA, querying its progress, and the rest of the functions described in this document. Other multichannel DMA registers, notably associated with interrupt handling, the API is completely uninvolved with and are strictly under the users' control.

This section lists multichannel DMA and multichannel DMA-related registers that are under the API's control versus those under the API caller's control. Defining the nature of these registers is beyond the scope of this document. More information about them is available in the appropriate chip specification.

The multichannel DMA registers are mapped into a structure defined in MCD_dma.h called dmaRegs.

2.8.1 Multichannel DMA Registers Under API's Control

The API controls the following registers of the multichannel DMA:

- TaskBAR
- CurrentPointer
- EndPointer
- VariablePointer
- Task Control registers
- Priority registers
- taskSize0 and taskSize1 registers
- Debug (the debug unit is used to provide DMA-pause function)
- PTD Control

2.8.2 Multichannel DMA Registers under API Caller's Control

User software must control the following registers of the multichannel DMA:

- Interrupt Pending register

- Interrupt Mask register
- Initiator Mux Control register

2.8.3 Multichannel DMA Register under User and API Control

The API uses the PTD debug registers in some cases to determine the status of a DMA when the function MCD_dmaStatus is called. The user may use these registers for debug but needs to be aware that the API may be accessing them also.

2.9 Misaligned Transfers

This API, in general, does not provide support for handling misaligned transfers. Misaligned transfers, as defined here, are those transfers where the source or destination address are not evenly divisible by the transfer size (xferSize) in bytes. Also, for most API call options, the total numbers of bytes must also be divisible by the transfer size in bytes.

3 API Function Calls

This section describes the API function calls.

Table 1. API Function Calls Summary

API Function Name	Description
MCD_initDma	Initialize the DMA API, and (optionally) copy the DMA microcode to on-chip SRAM for improved performance
MCD_startDma	Start a DMA operation
MCD_dmaStatus	Query the state of a DMA operation, e.g., completed, running, paused
MCD_XferProgrQuery	Query how far a DMA operation has gone so far
MCD_killDma	Stop a DMA operation
MCD_continDma	Continue a DMA operation if it has temporarily ended because of hitting a non-ready buffer descriptor; no effect if it has not ended
MCD_pauseDma	Pause a DMA in progress
MCD_resumeDma	Resume a paused DMA
MCD_csumQuery	Return a resulting checksum for an unchained DMA
MCD_getSize	Return size of DMA task code
MCD_getVersion	Returns version of DMA API

3.1 MCD_initDma()

```
int MCD_initDma (
      dmaRegs *dmaBarAddr, /* address of the MultiChannel DMA registers */
      void *microcodeDest,/* where to copy MultiChannel DMA microcode, usually in SRAM */
```

```
    u32 flags               /* initialization options */

    );
```

MCD_initDma() initializes the API and the DMA-controller registers that are under the API's control. It also terminates any DMAs that may be in progress, and can copy the DMA controller's "world" of tables and descriptors to the on-chip SRAM. This function should be called once during system initialization, before performing any other API calls.

The parameter, dmaBarAddr, informs the API where the multichannel DMA's memory-mapped registers are located. This is generally not the base address of the entire set of on-chip registers, but the address of the multichannel DMA registers in particular. This address is usually within the map of the entire set of on-chip registers. The data structure, dmaRegs, that matches the structure of the multichannel DMA registers, is defined in MCD_dma.h.

The address specified in the parameter microcodeDest, which should be in on-chip SRAM, tells MCD_initDma() where to move DMA tasks and associated structures to. This address must be on a 512-byte boundary.

The flags parameter specifies options for the MCD_initDma() call, as specified by #defined constants in MCD_dma.h:

- **MCD_RELOC_TASKS** to copy the microcode "tasks" to the specified place in SRAM. If this flag is not used, the user must take care to properly align the tasks and data structures in MCD_tasks.c in their linking process
- **MCD_NO_RELOC_TASKS** to leave the microcode where it is
- **MCD_COMM_PREFETCH_EN** (MCF547x/MCF548x only - should not be set for MPC8220 or unexpected operation could occur) to enable Comm bus prefetching

The flags parameter to the MCD_initDma() call is not related to the same-named parameter of the MCD_startDma() call.

MCD_initDma can return the following values:

- **MCD_TABLE_UNALIGNED** when the microcodeDest pointer is not aligned to 512 bytes or if any of the tables which need to be aligned (task table, function descriptor table, or variable table) are not aligned when not moving the tasks
- **MCD_OK** if there were no errors

3.2 MCD_startDma()

The most basic call in the DMA API is to start a DMA.

```
int MCD_startDma (
        int channel,   /* the channel on which to run the DMA */
        s8  *srcAddr,  /* the address to move data from, or buffer-descriptor address */
        s16 srcIncr,   /* the amount to increment the source address per transfer */
        s8  *destAddr, /* the address to move data to */
```

```
s16 destIncr,  /* the amount to increment the destination address per transfer */
u32 dmaSize,   /* the number of bytes to transfer independent of the transfer
     size */
u32 xferSize,  /* the number bytes in of each data movement (1, 2, or 4) */
u32 initiator, /* what device initiates the DMA */
int priority,  /* priority of the DMA */
u32 flags,     /* flags describing the DMA (explained below) */
u32 funcDesc   /* a description of byte swapping, bit swapping, and CRC actions */
```

The srcAddr parameter is defined differently depending on whether the flags parameters indicate that a single- or multiple-buffer DMA is requested. For single-buffer DMA, the srcAddr indicates the physical address of the source of the data. This can either be the address of the first piece of data in a data buffer or the address of a FIFO. For multiple-buffer tasks, the srcAddr parameter indicates the physical address of the first buffer descriptor in the chain or ring.

The destAddr parameter is also defined differently depending on the value of the flags parameter. For single-buffer DMAs, the destAddr indicates the physical address of the destination of the data. This can either be the address of the first piece of data in a destination buffer or the address of a FIFO. For multiple buffer chained DMAs, the destAddr has no meaning; source and destination addresses are defined entirely by each buffer descriptor. For the Ethernet DMAs, the destAddr holds the address of the transmit or receive FIFO. For Ethernet transmit, this truly is the destination address. For Ethernet receive, it is actually the source address.

For all types of DMA tasks, the srcIncr and destIncr indicate the amount to increment the actual source and destination addresses, whether they are defined by the srcAddr and destAddr parameters of MCD_startDma or by a buffer descriptor. Valid values must be passed for these arguments. If either the source or destination is a FIFO, the associated increment must be zero.

The dmaSize parameter is used in single-buffer DMAs to indicate the maximum frame length. It indicates the maximum number of bytes to transfer for that buffer.

The xferSize parameter indicates the width to transfer data in, such as byte, word (16 bit), or longword (32 bit). There are important interrelationships that must be observed between srcAddr, srcIncr, destAddr, destIncr, dmaSize, and xferSize parameters for single-buffer and multi-buffer chained DMAs. The amount of data to be transferred in the buffer must be evenly divisible by the transfer size. The Ethernet DMAs do not use this parameter and transfer all data except ending bytes in 4-byte chunks.

The initiator parameter must be supplied from a list of #defined initiator names for each peripheral device in a header file. These values are specific to the particular chip, and thus are generally named after the particular chip. For peripheral-to-memory or memory-to-peripheral DMAs, this should indicate the peripheral involved in that DMA. For memory-to-memory DMAs, it is preferable for this to indicate a timer used to limit DMA bandwidth. The Always initiator may also be used, if it is permissible to consume all available DMA capability. The user must set up the Initiator Mux Control register to ensure that the initiator is coming from the correct source.

The priority parameter is a value between 0 and 7 (inclusive), 7 meaning the highest priority. Higher-priority DMAs always take precedence over lower-priority DMAs, but DMAs do not compete for execution on this priority scale unless they are initiated by the initiator specified by the initiator parameter (i.e., unless that initiator is asserted).

The constants to be bit-wise ORed together to form the flags parameter are similar to those for the flags member of the chained buffer-descriptor format, described in Section 2.5.1, “Format for Chained DMAs”. Note that, unlike most of the constants ORed to form the funcDesc parameter, these options are selected by using them in the ORed expression, or deselected by leaving them out of that ORed expression. This is the case because they are strictly "true/false" questions rather than "multiple choice".

- **MCD_SINGLE_DMA** to cause the DMA channel to perform a single-buffer DMA described completely by the parameters of the call. If MCD_SINGLE_DMA is not supplied in the ORed expression, then the DMA is interpreted as a chained, buffer-descriptor-based DMA, and the source address (srcAddr) parameter points to the first buffer descriptor in the chain.
- **MCD_FECTX_DMA** to cause the DMA to use FEC transmit task. If this flag is set, it takes priority over MCD_SINGLE_DMA.
- **MCD_FECRX_DMA** to cause the DMA to use FEC receive task. If this flag is set, it takes priority over MCD_SINGLE_DMA.
- **MCD_INTERRUPT** to cause the DMA channel to generate an interrupt after completion of the DMA.
- **MCD_END_FRAME** to instruct the DMA to terminate the frame, packet, cell, etc. on the transmitted data stream. If the DMA is not transmitting data, or if the peripheral involved does not deal with frames, packets, or the like, then this has no effect.
- **MCD_CRC_RESTART** to empty out the accumulated checksum prior to performing the DMA. This flag only affects single buffer DMAs when MCD_INCLUDE_EU is defined and the funcDesc parameter instructs the API to perform some sort of data manipulation. Unlike in chained DMAs where the resulting CRC is in a member of the buffer descriptor, in a single-buffer DMA as specified by the function parameters, the resulting checksum can be retrieved using the MCD_csumQuery() function call
- **MCD_TT_FLAGS_CW** instructs the API to set the task’s write combine bit.
- **MCD_TT_FLAGS_RL** instructs the API to set the task’s read line bit.
- **MCD_TT_FLAGS_SP** (MCF547x/MCF548x only - should not be set for MPC8220 or unexpected operation could occur) instructs the API to set the task’s speculative prefetch bit.

Typically, for the MCD_TT_FLAGS flags, write combine and read line should be set. If accessing a FIFO outside of the communications subsystem such as on the FlexBus, the relevant MCD_TT_FLAG should not be set.

The funcDesc parameter controls whether the data being transferred by the DMA is modified or operated on in any way. Most of the parameters to this argument are only valid if the compile time option MCD_INCLUDE_EU is defined. If this constant is not defined, no operations will be performed on the data as it is transferred.

The parameter funcDesc are specified by #defined constants bit-wise ORed together, those constants being supplied in a MCD_dma.h file. Those constants for funcDesc are the following:

- Byte swapping:
  - **MCD_NO_BYTE_SWAP** to disable byte swapping.
  - **MCD_BYTE_REVERSE** to reverse the bytes of each u32 of the DMAed data. The high-order byte is exchanged with the low-order, and the second-highest-order byte is exchanged with the second-lowest-order byte. This is an endian change for 32-bit quantities.
  - **MCD_U16_REVERSE** to reverse the 16-bit halves of each 32-bit data value being DMAed.
  - **MCD_U16_BYTE_REVERSE** to reverse the byte halves of each 16-bit half of each 32-bit data value DMAed, preserving the order of the 16-bit halves. This is an endian change for 16-bit quantities. Exactly one of these constants must appear in the ORed expression for funcDesc.

Leaving out MCD_NO_BYTE_SWAP when no byte swapping is desired will not turn off byte swapping. Byte swapping, if selected, is always performed on 32-bit quantities and on 32-bit boundaries. If the transfer size is other than 32-bit (4), then the resulting behavior is not defined.

- Bit reversing:
  - **MCD_NO_BIT_REV** do not reverse the bits of each byte DMAed.
  - **MCD_BIT_REV** reverse the bits of each byte DMAed (bit 7 exchanges with bit 0, bit 6 exchanges with bit 1, bit 5 with bit 2, etc.).

It is recommend supplying one or the other of these two constants in the ORed expression for funcDesc, but if neither is supplied, no bit swapping will occur.

- CRCing:
  - **MCD_CRC16** to perform CRC-16 on DMAed data.
  - **MCD_CRCCCITT** to perform CRC-CCITT on DMAed data.
  - **MCD_CRC32** to perform CRC-32 on DMAed data.
  - **MCD_CSUMINET** to perform internet (IP) checksums on DMAed data.
  - **MCD_NO_CSUM** to perform no checksumming.

Exactly one of these constants must be supplied in the ORed value for funcDesc. Leaving it out of the ORed expression is not synonymous with MCD_NO_CSUM. The CRCs implement the polynomial for the CRC but do not allow for variations on the algorithm, such as initializing the CRC with a value besides zero or XORing the resulting CRC with a user supplied value. In addition, the IP checksum does not execute the final complement of the checksum required by the IP checksumming algorithm. That must be done in software.

MCD_startDma can return the following values:

- **MCD_CHANNEL_INVALID** when the channel parameter is not valid (not within 0-15)
- **MCD_OK** if there are no errors

3.3 MCD_dmaStatus

```
int MCD_dmaStatus (
```

```
      int channel   /* the DMA channel whose status to retrieve */
);
```

MCD_dmaStatus() returns the status of the DMA on the requested channel. The status is defined by these constants from MCD_dma.h:

- **MCD_NO_DMA** if no DMA has ever been requested for this channel since the API was initialized
- **MCD_IDLE** DMA active for this channel, but the DMA initiator is currently inactive
- **MCD_RUNNING** DMA active for this channel, and transfers are happening
- **MCD_PAUSED** DMA active for this channel, but is currently paused
- **MCD_HALTED** the most recent DMA on this channel has been killed by the CPU with MCD_killDma()
- **MCD_DONE** the most recent DMA on this channel has been completed. This could occur upon completion of a single-buffer DMA as defined by the function parameters, or when a chained DMA encounters a buffer descriptor that is not ready (bufferDesc.flags & MCD_BUF_READY == 0).
- **MCD_CHANNEL_INVALID** when the channel parameter is not valid (not within 0-15)

3.4 MCD_XferProgrQuery()

```
int MCD_XferProgrQuery (
      int channel,              /* the DMA channel for which to query progress */
      MCD_XferProg *progRep     /* structure pointer detailing DMA progress */
);
```

MCD_XferProgrQuery() queries the progress of a DMA. The MCD_XferProg structure, defined in MCD_dma.h, provides the following information:

```
typedef struct MCD_XferProg_struct {
      s8 *lastSrcAddr; /* the most recent or last, post-increment source address */
      s8 *lastDestAddr; /* the most recent or last, post-increment destination address */
      u32 dmaSize; /* the amount of data transferred for the current buffer */
      MCD_bufDesc *currBufDesc; /* pointer to the current buffer descriptor being
                                                                        DMAed */
} MCD_XferProg;
```

Upon completing or after aborting a DMA with MCD_killDma(), or while the DMA is in progress, this function returns the first DMA-destination address not (or not yet) used in the DMA. The member currBufDesc of the MCD_XferProg structure has no meaning in single-buffer DMAs.

Progress-query information begins with an initial value of the first addresses of the DMA (in the first buffer descriptor in multi-buffer DMAs), and with a zero dmaSize. As transfers are performed, this information

is periodically, but not continuously, updated. This information is not updated while not transferring data (such as while parsing buffer descriptors), since only during transfers is there any DMA progress to be reported. If a buffer descriptor does not result in any data transfers being performed, such as if that buffer descriptor is not ready, they similarly do not produce any DMA progress to be reported. Progress-query results are nevertheless valid anytime after a DMA is started on a given channel.

Calling MCD_XferProgrQuery() for a chained DMA that has already completed will report that the requested number of bytes were transferred in the last buffer descriptor that transferred any data. It will not show zero bytes transferred in the non-ready buffer descriptor that stopped the DMA, because the non-ready buffer descriptor did not produce DMA progress to report. Showing zero bytes transferred in the non-ready buffer would overwrite useful information about the DMA transfers, such as where the last transfers were performed. Reporting zero bytes transferred in the non-ready, terminating buffer is also inconsistent with progress reporting for single-buffer DMAs, whereas reporting how much was transferred in the last active buffer matches single-buffer DMA results.

MCD_XferProgrQuery can return the following values:

- **MCD_CHANNEL_INVALID** when the channel parameter is not valid (not within 0-15)
- **MCD_OK** if there were no errors

3.5 MCD_killDma

```
int MCD_killDma (
      int channel    /* channel whose DMA to is to be aborted */
);
```

MCD_killDma() terminates the DMA task running on the channel specified by the parameter. A DMA may be killed from any state, including paused, and it always goes to the MCD_HALTED state even if it is killed while in the MCD_NO_DMA or MCD_IDLE states.

MCD_killDma can return the following values:

- **MCD_CHANNEL_INVALID** when the channel parameter is not valid (not within 0-15)
- **MCD_OK** if there were no errors

3.6 MCD_contInDma()

```
int MCD_contInDma (
      int channel    /* channel whose DMA is to be continued */
);
```

MCD_contInDma() continues a DMA that has already been started but may have stopped. The intended use of this function is to restart a multi-buffer DMA that has stopped due to encountering an unready buffer descriptor. Once the buffer descriptor is marked ready, MCD_contInDma can be called and the DMA will resume operation at that buffer descriptor. This call may also be used to restart a DMA after inserting or removing buffer descriptors from a chain.

MCD_contInDma can return the following values:

- **MCD_CHANNEL_INVALID** when the channel parameter is not valid (not within 0-15)
- **MCD_OK** if there were no errors

3.7 MCD_pauseDma()

```
int MCD_pauseDma (
      int channel       /* the DMA channel to pause */
);
```

This function is used to pause the DMA on a particular channel. Only one channel can be paused at a time. The multichannel DMA debug unit is used to match the channel number which is passed and block the initiator of that task. Pausing a channel that is already paused has no effect.

MCD_pauseDma can return the following values:

- **MCD_CHANNEL_INVALID** when the channel parameter is not valid (not within 0-15)
- **MCD_OK** if there were no errors

3.8 MCD_resumeDma()

```
int MCD_resumeDma (
      int channel       /* the DMA channel to resume */
);
```

This function is used to resume a DMA which has been paused. Resuming a channel that is not paused may produce unpredictable results.

MCD_resumeDma can return the following values:

- **MCD_CHANNEL_INVALID** when the channel parameter is not valid (not within 0-15)
- **MCD_OK** if there were no errors

3.9 MCD_csumQuery()

```
int MCD_csumQuery (
      int channel, /* the channel whose checksum to query */
      u32 *csum /* pointer to where to store the checksum */
);
```

MCD_csumQuery() returns the checksum value after performing a non-chained DMA. For chained DMAs, the checksum is provided in the buffer descriptor. This function is only useful when MCD_INCLUDE_EU is defined.

MCD_csumQuery can return the following values:

- **MCD_CHANNEL_INVALID** when the channel parameter is not valid (not within 0-15)
- **MCD_OK** if there were no errors

3.10 MCD_getVersion()

```
int MCD_getVersion (
      char **longVersion /* pointer to a longer version name and build date*/
);
```

This returns a numeric major and minor revision number in the two least significant bytes of the return value. Revision 1.1 would be indicated as 0x00000101. This also returns a pointer to a string in the longVersion parameter. This can be referenced for more information. An example would be:

```
      “Multichannel DMA API alpha v.0.1 120304”
```

3.11 MCD_getSize()

```
int MCD_getSize ();
```

This returns the packed size of the image which can be used to determine how much space needs to be allocated for the DMA code and structures. This value will typically be less when MCD_INCLUDE_EU is not defined.

4 Example Code

The following sections contain example code for using the multichannel DMA API.

4.1 Initialization

The following code shows an example call for MCD_initDma which relocates the API microcode and structures to the on-chip SRAM.

```
#include “MCD_dma.h”
#define MBAR_BASE_ADRS 0xf0000000 /* base address of chip internal regs */
#define MMAP_DMA 0x00008000 /* offset of DMA regs from register base */
#define MMAP_SRAM 0x00020000 /* offset of on-chip SRAM from register base */


MCD_initDma(
      (dma_regs*) (MBAR_BASE_ADRS + MMAP_DMA),
      (void*) (MBAR_BASE_ADRS + MMAP_SRAM),
      (MCD_RELOC_TASKS)
```

```
);
```

On MCF547x/548x parts, the MCD_COMM_PREFETCH_EN flag can be ORed in with the MCD_RELOC_TASKS flag.

HOW TO REACH US:

USA/Europe/Locations not listed:
Freescale Semiconductor Literature Distribution
P.O. Box 5405, Denver, Colorado 80217
1-800-521-6274 or 480-768-2130

Japan:
Freescale Semiconductor Japan Ltd.
Technical Information Center
3-20-1, Minami-Azabu, Minato-ku
Tokyo 106-8573, Japan
81-3-3440-3569

Asia/Pacific:
Freescale Semiconductor H.K. Ltd.
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T. Hong Kong
852-26668334

Learn More:
For more information about Freescale Semiconductor products, please visit
http://www.freescale.com

MCDMAAPIUG
Rev. 0, 7/2004

• Preliminary

freescale™ semiconductor