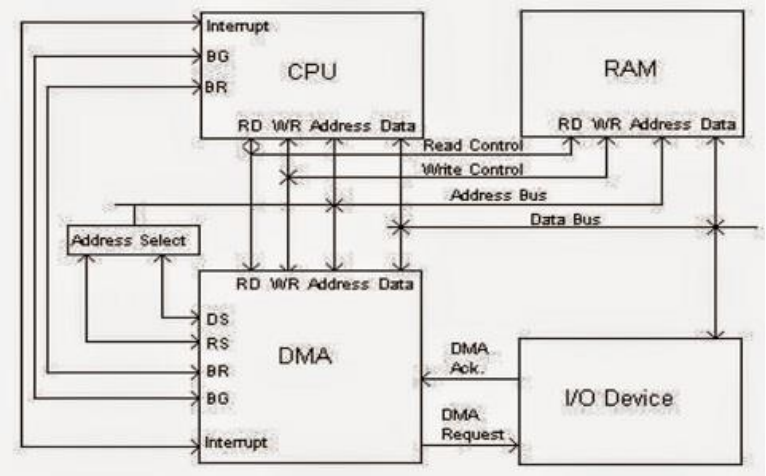
Note: register addresses specific to BCM2836/2837

Raspberry Pi DMA GPIO

Version 2.0

DMA Controllers

The BCM2837 has a hardware device called a direct memory access (DMA) controller responsible for data transfers between the processor, memory, and I/O devices. Placing this hardware device between the processor/memory and I/O devices allows the processor itself to no longer be responsible for data transfer thus freeing it up to do other tasks. A DMA controller is included in pretty much every computer for this reason.



DMA controllers, while transferring data without the intervention of the processor, is controlled by the processor. The processor instructs the DMA on memory addresses, amount of data, and the direction of the transfer (from memory to I/O devices and vice versa). The DMA controller then completes the task and notifies the processor by raising an interrupt. When an I/O device wants to initiate a transfer, it notifies the DMA controller itself. The DMA controller will then request the processor for the data bus and then makes the transfer when able.

The BCM2837 makes use of it's DMA controller to accomplish these tasks, but since the majority of hardware pipelines and peripherals are bus masters (the device is able to initiate DMA transactions without the need of a dedicated DMA controller, "first-party DMA"), the controller mainly support some of the simpler peripherals.

Using the DMA Controller on the Pi

The DCM2837 DMA controller (pg. 38 of the peripheral reference) provides a total of 16 DMA channels operating independently from each other and are internally arbitrated onto on the 3 system busses. Each one of these channels operates by loading a control block (CB) data structure from memory into internal registers. Each one of these control blocks can point to a further control block to be loaded and executed once the operation described in the current control block (e.g. setting or clearing a GPIO pin) has been completed. In effect, a linked list of CBs can be constructed in order to execute a sequence of DMA operations without software intervention.

The DMA Channel register sets are located by an offset to base addresses of DMA Channel 0 (0x7E007100) and DMA Channel 15 (0x7EE05000). Each DMA channel has an identical register map, i.e. only the base address of said channels are different. Note, only three registers in each channel are directly writeable (CS, CONBLK_AD, and DEBUG); the other registers (TI, SOURCE_AD, DEST_AD, TXFR_LEN, STRIDE & NEXTCONBK) are automatically loaded from a Control Block data structure held in external memory.

Offset to DMA Channel 0	DMA Channel	Offset to DMA Channel 0	DMA Channel
0x000	0	0x800	8
0x100	1	0x900	9
0x200	2	0xa00	10
0x300	3	0xb00	11
0x400	4	0xc00	12
0x500	5	0xd00	13
0x600	6	0xe00	14
0x700	7		

Offset to DMA Channel 15	DMA Channel
0x000	15

The global enable register at the top of each channel set can disable each DMA for power saving. For the Raspian OS, DMA channels 0, 1, 2, 3, 6, and 7 are typically in use by other processes.

Register Map

The DMA controller register map for an arbitrary channel. Offset for specific channels can be found above.

Address Offset from DMA + Channel	Register Name	Description	Size
0x0	#_CS	Control and Status	32
0x4	#_CONBLK_AD	Control Block Address	32
0x8	#_TI	Transfer Information	32
0xc	#_SOURCE_AD	Source Address	32
0x10	#_DEST_AD	Destination Address	32
0x14	#_TXFR_LEN	Transfer Length	32
0x18	#_STRIDE	2D Stride	32
0x1c	#_NEXTCONBK	Next CB Address	32
0x20	#_DEBUG	Debug	32

Details on each register are found on page 47 of the peripheral handbook.

Control Blocks

CBs are 8 words (256 bits) in length and must start at a 256-bit aligned address. Each 32-bit word of the control block is automatically loaded in the corresponding 32-bit DMA control block register (#_CONBLK_AD) at the start of the DMA transfer. The descriptions of these registers also define the corresponding bit locations in the CB data structure in memory (i.e. it's a one-to-one mapping).

32-bit Word Offset	Description	Associated Read-Only Register
0	Transfer Information	TI
1	Source Address	SOURCE_AD
2	Destination Address	DEST_AD
3	Transfer Length	TXFR_LEN
4	2D Mode Stride	STRIDE
5	Next Control Block Address	NEXTCONBK
6	Reserved (set to zero)	N/A
7		

The DMA is started by writing the address of a CB structure into the CONBLK_AD register and then setting the ACTIVE bit. The DMA will fetch the CB from the address set in the SCB_ADDR of this register and it will load it into the read-only registers TI, SOURCE_AD, DEST_AD, TXFR_LEN, STRIDE, and NEXTCONBK. It will then begin a DMA transfer according the information in the CB.

When the DMA transfer is complete, the DMA will update the CONBLK_AD register with the contents of the NEXTCONBK register. A new CB will be fetched using this address and the procedure starts again. This process only stops when the NEXTCONBK register is set to 0x0000_0000. This address is loaded into CONBLK_AD register and the DMA controller stops.

The majority of registers are read-only as their contents are automatically populated using the CB. The value loaded into the NEXTCONBK register can be overwritten so that the linked list of Control Block data structures can be dynamically altered. However, it is only safe to do this when the DMA is paused.

PWM and the DMA Controller

PWM Controller

The Pi has a dedicate PWM controller (base address of 0x7E20C000) that incorporates

- Two independent output bit-streams (PWM0 and PWM1), clocked at a fixed frequency.
- PWM outputs that have variable input and output resolutions.
- Two modes: PWM or serialized version of 21-bit words
    - PWM: Utilize the M-N algorithm described on page 139 to transmit data stored in either DATi or FIF1 buffer via PWM. Default mode.
    - Serialized: Transmit data stored in either DATi or FIFi buffer serially.
- Clock by PWM clock manger (CM_PWMCLK)

Summary description of the PWM controller registers. Full descriptions are found on page 142.

Address Offset	Register Name	Description	Size
0x0	CTL	PWM Control (mode and channel control)	32
0x4	STA	PWM Status	32
0x8	DMAC	PWM DMA Configuration	32
0x10	RNG1	PWM Channel 1 Range (pulse width)	32
0x14	DAT1	PWM Channel 1 Data (data to output if CTL USEFi = 0)	32
0x18	FIF1	PWM FIFO Inputer (data to output if CTL USEFi = 1)	32
0x20	RNG2	PWM Channel 1 Range (pulse width)	32
0x24	DAT2	PWM Channel 1 Data (data to output if CTL USEFi = 0)	32

The two output channels, PWM0 and PWM1, are assigned as alternative functions to 4 GPIO pins. This means that only two PWM outputs can be achieved on the Pi at a given instance in time.

GPIO	PWM0	PWM1
12	Alternative Function 0	
13		Alternative Function 0
18	Alternative Function 5	
19		Alternative Function 5

PWM Clock Manager

The PWM controller relies upon a dedicated clock generator called CM_PWMCTL (base address of 0x7E101000) for clocking data output. This clock manager produces an even duty cycle configured by choosing a clock source and integer divisor set by reading from CM_PWMCTL and CM_PWMDIV. PWM controller uses this clock source to time its data output; in PWM or serial mode, the register RNGi defines the period of length (number of clock cycles) in which data is transmitted in. Specifically:

1. PWM: Evenly distributed pulses are sent within a period of length defined in RNGi.
2. Serial: Data is transmitted within the same period of length defined in RNGi.

Summary of the PWM clock manager registers. Full descriptions found in addendum.

Address Offset	Register Name	Description	Size
0xa0	CM_PWMCTL	PWM Clock Manager Control (clock source + enable)	32
0xa4	CM_PWMDIV	PWM Clock Manager Divisors (12 bits for integer, 12 bits for fractional)	32

The clock sources available for use and selectable within CM_PWMCTL:

CM_PWMCTL Bit 3-0 Value	Clock Source	Frequency
0, 8 – 15	Ground	0 Hz
1	Oscillator	19.2 MHz
2	Testdebug0	0 Hz
3	Testdebug1	0 Hz
4	PLLA	0 Hz
5	PLLC	1 GHz
6	PLLD	500 MHz
7	HDMI auxiliary	216 MHz

Ultimately, only two useful clock sources that can be used: 19.2 MHz oscillator and 500 MHz PLLD (phase-locked loop). The integer divisor, having a length of 12 bits, means that either clock source can be at most divided by 4095. This leaves the minimum frequency achievable to be 122 kHz and 4.7 kHz for the oscillator and PLLD source.

The PWM controller has a very precise and know delay when it writes out data from DATi of FIF1. This delay, referred to as the pulse width, is determined by the clock frequency and value of RNGi.

$$\Delta t_{pulse} = \frac{RNGi}{f_{clock}}$$

For example, consider the PLLD clock is divided to 10 MHz. To achieve a pulse duration of 10µs, range must be set to 100 cycles. Thus, the PWM controller will be writing out data for a pulse duration of 10µs.

DMA Controller to Create PWM

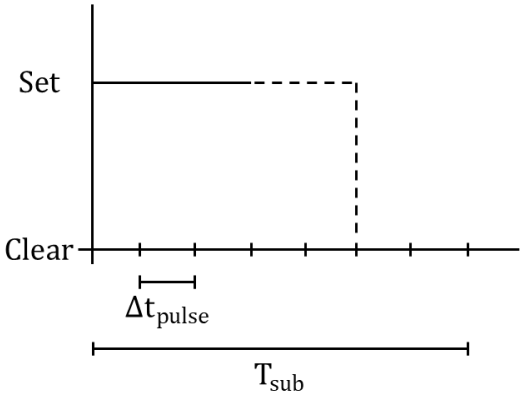
The basic idea is this: the PWM controller has a very precise and known delay when it writes out data placed in its DATi or FIF1 buffer. We can use the DMA controller to write data to this location and leverage that very well-known delay to allow us to set and clear GPIOs at a specific cadence. In turn, we can create a PWM output on any GPIO pin we want. A sequence of control blocks in a circular linked list will be used to achieve a desired PWM signal.

An arbitrary control block chain with a pulse width of 10µs will have the following order to set and clear a GPIO pin:

CB Number	Source Data	Destination Address	Elapsed Time
1	Set GPIO Pin Mask	GPSETn	0µs
2	Any Data	FIF1	0µs
3	Any Data	FIF1	10µs
n – 1	Any Data	FIF1	(n – 2) * 10µs
n – 2	Any Data	FIF1	(n – 1) * 10µs
n	Clear GPIO Pin Mask	GPCLRn	(n – 1) * 10µs

To create a block chain built based upon an input frequency and duty cycle, will need to define a period in which this sequence repeats in. This is important as our sequence is finite in length where the last block is linked to the first one. We will define an overall cycle T equal to 1 second and then a variable sub cycle T_{sub}. T_{sub} will equal the desired PWM signal frequency as this sub cycle will be repeated for 1 second:

$$f_{PWM} = \frac{T}{T_{sub}} = \frac{1}{T_{sub}}$$



The number of control blocks within the sequence repeated every T_{sub} will be a function of the pulse width Δt_{pulse} as our sequence is clocked by the PWM controller. Excluding the two control blocks to set and clear the GPIO pins, the number of CB will then be:

$$N_{CB} = \text{floor}\left(\frac{T_{sub}}{\Delta t_{pulse}} * \frac{1}{2}\right)$$

Consequently, the number of pulses, or CBs, elapse while a GPIO is set and cleared is defined by the input duty cycle:

$$N_{pulses,high} = N_{CB} * \text{Duty}$$

$$N_{pulses,low} = N_{CB} * (1 - \text{Duty})$$

A few things to note:

- The number of control blocks must be an integer. This means that in some cases, the desired frequency of the PWM signal cannot be met exactly. I choose to floor this number.
- Duty cycles of 100% or 0% are special cases. The number of pulses the GPIO is high or low remains the same, but the control block responsible for setting or clearing the GPIO need to be removed.

Determining the pulse width is a function of the desired resolution of the PWM duty cycle and maximum allowed number of control blocks. For example, if the desired frequency is 18 kHz and Δt_{pulse} is set to 10µs (reference clock = 10 MHz and PWM RNG1 = 100):

$$18\ \text{kHz} = \frac{1}{T_{sub}} \therefore\ T_{sub} = 55.56\ \mu s$$

$$N_{CB} = \text{floor}\left(\frac{T_{sub}}{\Delta t_{pulse}} * \frac{1}{2}\right) = 2$$

$$\text{Duty Resolution} = 100 * \left(1 - \frac{(N_{CB} - 1)}{N_{CB}}\right) = 50\%$$

Any input duty cycle must then round to the nearest 20%. Note that the desired frequency of 18 kHz is not met as the number of control blocks in the sequence was rounded down to 5; thus, an actual frequency of 20 kHz is produced. To move around this limitation, need to decrease the pulse width. The table below gives further examples of this calculation for a wide range of input frequencies for a set clock divisor and PWM RNG1 value.

clock	500	Mhz
divisor	50	

clock	10	Mhz
pwm rng	100	

pulse width	1.00E-05	s
	10	us

Input Freq		T_sub		N_cb (floored)	Duty res	memory size		pages	act freq	
0.05	hz	20.0000000	s	1000000	0.00%	32000000	B	7813	0.1	Hz
0.1	hz	10.0000000	s	500000	0.00%	16000000	B	3907	0.2	Hz
1	hz	1.0000000	s	50000	0.00%	1600000	B	391	2	Hz
10	hz	0.1000000	s	5000	0.02%	160000	B	40	20	Hz
50	hz	0.0200000	s	1000	0.10%	32000	B	8	100	Hz
100	hz	0.0100000	s	500	0.20%	16000	B	4	200	Hz
1000	hz	0.0010000	s	50	2.00%	1600	B	1	2000	Hz
5000	hz	0.0002000	s	10	10.00%	320	B	1	10000	Hz
10000	hz	0.0001000	s	5	20.00%	160	B	1	20000	Hz
18000	hz	0.0000556	s	2	50.00%	64	B	1	50000	Hz
20000	hz	0.0000500	s	2	50.00%	64	B	1	50000	Hz

Implementation

Implementing hard PWM via the DMA controller is split into these sections:

1. Memory Management
2. Hardware configuration
3. Control block sequence
4. Configuring and Starting DMA
5. Halting DMA

Memory Management

Memory usage is the most critical aspect of implementing PWM via DMA. This is because the DMA controller insists upon uncached 256-bit aligned memory bus addresses for each control block, source address, and destination. Because we work in possibly cached virtual memory, there needs to be a serious discussion on how we satisfy this requirement.

Firstly, uncached memory, as opposed to cached memory, describes memory held in main memory and not in the processor's L1, L2, etc. cache buffers. These L1, L2, etc. caches hold frequently requested data and instructions so that they are immediately available to the CPU when needed thus decreasing the average cost to access said data; they are buffering data between the CPU and RAM. Unfortunately, the

DMA controller is unaware of the caches (as they are not directly accessible over the peripheral bus) meaning that any memory located in these caches will not be available to it. Our process will be running in userspace meaning that any memory we allocate has a very good chance of presiding in these cache buffers. We must then take the extra step to allocate uncached memory and use that block of memory for any data accessed by the DMA controller.

Allocating uncached memory is reserved for kernel space applications where specific functions, such as `dma_alloc_coherent()`, are available to you. If you want to keep everything in userspace, then we will have to fall back on interfacing with character devices that have this functionality available for us to use. Broadcom thankfully provide what is called a "mailbox" interface with the VideoCore graphics processor that we can use to allocate aligned uncached memory from userspace. This mailbox interface provides a wide range of functions beyond allocating memory and can be learned more about here.

The mailbox interface is collection of these functions:

- mbox_open()
  - Open device /dev/vcio (VideoCore interface character device)
- mbox_close(int file_desc)
  - Close device /dev/vcio (VideoCore interface character device)
- mem_alloc(int file_desc, unsigned size, unsigned align, unsigned flags)
  - Allocate memory with specific requirements (coherent, initialized, uncached, etc)
  - Flags:

```
MEM_FLAG_DISCARDABLE     (1 << 0)  // Can be resized to 0 at any time. Use for cached data
MEM_FLAG_NORMAL          (0 << 2)  // Normal allocating alias. Don't use from ARM
MEM_FLAG_DIRECT          (1 << 2)  // 0xC alias uncached
MEM_FLAG_COHERENT        (2 << 2)  // 0x8 alias. Non-allocating in L2 but coherent
MEM_FLAG_ZERO            (1 << 4)  // Initialize buffer to all zeros
MEM_FLAG_NO_INIT         (1 << 5)  // No initialization (default is initialise to all ones)
MEM_FLAG_HINT_PERMALOCK  (1 << 6)  // Likely to be locked for long periods of time
MEM_FLAG_L1_NONALLOCATING \
    (MEM_FLAG_DIRECT | MEM_FLAG_COHERENT)  // Allocating in L2
```

- mem_free(int file_desc, unsigned handle)
  - Free allocated memory
- mem_lock(int file_desc, unsigned handle)
  - Lock allocated memory to RAM so it can be accessed
- mem_unlock(int file_desc, unsigned handle)
  - Unlock allocated memory in RAM so it can be freed
- mapmem(unsigned base, unsigned size)
  - Map allocated memory into calling process (get virtual address)
- unmapmem(void *addr, unsigned size)
  - Unmap allocated memory from calling process

The procedure to allocate and free uncached memory will be to

```
// Allocate uncached memory:
uncached_malloc(size, alignment) {
    // Open mailbox
    fd = mbox_open();

    // Allocate memory
    mb_handle = \
        mem_alloc(fd, size, alignment, MEM_FLAG_L1_NONALLOCATING);

    // Lock memory
    bus_addr = mem_lock(fd, mb_handle);

    // Map memory
    virt_addr = \
        mapmem(bus_addr & ~0xC0000000), size);

    // Close mailbox
    mbox_close(fd);
}

// Free allocated uncached memory
uncached_free() {
     // Open mailbox
    fd = mbox_open();

    // Unmap memory
    unmapmem(virt_addr, size);

    // Unlock memory
    mem_unlock(fd, mb_handle);

    // Free memory
    mem_free(fd, mb_handle);

    // Close mailbox
    mbox_close(fd);

}
```

Now that we have a function to allocate uncached memory, let's now focus on satisfying the remaining two requirements the DMA controller imposes upon us: 256-bit aligned bus addresses. Two requirements are imbedded in this statement as we must pass not only control blocks living in 256-bit aligned memory chunks but bus addresses (not virtual) too.

1. The DMA controller insists upon 256-bit aligned bus addresses for each control block because it loads 32 bytes of data starting from the addresses it's given and assumes that a control block is contained within. This requirement is satisfied by passing any alignment to uncached_malloc() as long as it's a multiple of 32 (256/8=32) but I opt to allocate memory at page aligned addresses (4096 bytes on most machines) to ensure that no control block spans a page boundary. Remember that memory addresses are said to be n-byte aligned when the address is a multiple of n bytes.
2. Determining bus memory addresses from virtual addresses is typically done through opening /proc/pid/pagemap to determine a physical address and then masking to find the bus address (if required). However, in allocating uncached memory, we already know the base bus address for our allocated region. Determining the bus address for any virtual memory address within this region is then trivial:

```
// Translate virtual to bus address of uncached memory
uintptr_t uncached_virt_to_bus_addr(void *ptr) {
    // Definitions:
    uintptr_t offset;

    // Find offset of address to its base:
    offset = (char*)ptr - (char*)virt_addr;

    // Check that offset falls within allocated memory size:
    if (offset <= size) {
        // Return base buss address plus offset:
        return (bus_addr + offset);
    } else {
        // Return with error
        return -1;
    }
}
```

Hardware Configuration

PWM via DMA controller relies upon the PWM controller and PWM clock manager. Ensuring a correct and predictable implementation means configuring each of these hardware devices in a specific way prior to loading the control block sequence and starting the engine.

PWM Controller & Clock Manager

The PWM controller and clock manager are to be initialized and configured prior to loading the control block sequence. Keep in mind that these configurations will apply to all DMA channels and any other process that uses the hardware devices (like audio). A delay between poking registers of 10 microseconds is required for some operations but recommended for all.

Step	Device	Register	Value	Purpose
1	PWM Controller	CTL	0x00	Reset
Delay				
2	PWM CM	PWMCTL	(0x5A << 24) | (6 << 0)	Set clock source to PLLD
Delay				
3	PWM CM	PWMDIV	(0x5A << 24) | (*X* << 12)	Set clock divisor
Delay				
4	PWM CM	PWMCTL	(0x5A << 24) | (6 << 0) | (1 << 4)	Enable clock
Delay				
5	PWM Controller	RNG1	*X*	Set period of length
Delay				
6	PWM Controller	DMAC	(1 << 31) | (15 << 0) | (15 << 8)	Enable DMA and set thresholds
Delay				
7	PWM Controller	CTL	(1 << 6)	Clear FIFO buffer
Delay				
8	PWM Controller	CTL	(1 << 5) | (1 << 0)	Use FIFO and enable channel

Data request (DRQ) and panic signal thresholds for the PWM controller's DMA configuration sets the threshold level for those two signals going active. We set them both to 15 (their max value) to force the PWM controller to only request more data from the DMA controller only when the FIFO buffer is almost empty. If these thresholds we set to a smaller value, then we would no longer get a well-known delay from the PWM controller.

Recall that the clock divisor and period of length is determined by a trade between the number of control blocks within the sequence (the amount of allocated memory required) and the target PWM signal frequency. See the spreadsheet example for recommendations on what to set these values to during initialization.

Control Block Sequence

A control block sequence needs to be constructed within the block of allocated uncached memory that produces a PWM signal of *N* duty cycle. Controlling the duty cycle is done by forming a sequence of N_{CB} long divided into $\mathrm{N}_{\mathrm{pulses,high}}$ and $\mathrm{N}_{\mathrm{pulses,low}}$ control blocks separated by a clear GPIOs if the duty cycle is not 100%. An additional control block will be prepended to this sequence to set or clear GPIOs depending upon the duty cycle.

Only a maximum of three different types of control blocks will have to be built for this sequence: set GPIO, clear GPIO, and wait (write to PWM controller). Below is a tabulation of what value will populate each control block field for the three different types:

Field	GPIO Set	GPIO Clear	Wait
TI	(1 << 26) | (1 << 3)	(1 << 26) | (1 << 3)	(1 << 26) | (1 << 3) | (1 << 6) | (5 << 16)
SOURCE_AD	Bus address of mask	Bus address of mask	Random bus address
DEST_AD	0x7E20001C	0x7E200028	0x7E20C018
TXFR_LEN	0x4	0x4	0x4
STRIDE	0x0	0x0	0x0
NEXTCONBK	Next CB Bus address*	Next CB Bus address*	Next CB Bus address*

*Bus address of the next control block unless the last control block of the sequence. In this case, link back to the first control block to form a circular linked list.

The GPIO transfer information field value is a minimum of two flags: no wide bursts (1 << 26) and wait for response (1 << 3). Setting these two flags ensures that the DMA controller will write data all at once and will wait for a response from the peripheral it wrote to. For a wait CB, add two additional flags: data required (1 << 6) and set mapped peripheral (5 << 16). The data required flag instructs the controller to gate any additional write outs until the selected peripheral raises a data required signal; the set mapped peripheral flag selects this gating peripheral (PWM controller = 5). See page 61 for a complete list of DREQ.

The algorithm to form this sequence of these control blocks is straight forward. The only special cases to be aware of is for duty cycles of 0% or 100% as this means that GPIOs will only ever have to be set or cleared.

```
// Build CB sequence:
for (i = 0; i < n_cb; i++) {
    // Set or clear GPIOs depending on duty cycle for first CB:
    if (i == 0) {
         // Build control block
         dma_cb_seq->info = DMA_NO_WIDE_BURSTS | DMA_WAIT_RESP;

         // Set GPIOs if duty cycle is not 0:
         if (duty_cycle != 0) {
             dma_cb_seq->src = set_mask_bus_addr;
             dma_cb_seq->dst = GPSET0_BUS_ADDR;
         } else {
             dma_cb_seq->src = clear_mask_bus_addr;
             dma_cb_seq->dst = GPCLR0_BUS_ADDR;
         }
         dma_cb_seq->length = 4;
         dma_cb_seq->stride = 0;
         dma_cb_seq->next = uncached_virt_to_bus_addr(cb_base, (dma_cb_seq + 1));
```

```
    // Clear GPIOs for non-trivial duty cycles (no need to clear in
    // those cases once "wait" CB while GPIO is set expends):
    } else if ((i == (n_pulses_high + 1)) && (duty_cycle % 100 != 0)) {
        // Build control block
        dma_cb_seq->info = DMA_NO_WIDE_BURSTS | DMA_WAIT_RESP;
        dma_cb_seq->src = clear_mask_bus_addr;
        dma_cb_seq->dst = GPCLR0_BUS_ADDR;
        dma_cb_seq->length = 4;
        dma_cb_seq->stride = 0;
        dma_cb_seq->next = uncached_virt_to_bus_addr(cb_base, (dma_cb_seq + 1));
     // Write data to PWM controller to wait if no need to clear or
     // set GPIOs:
     } else {
        // Build control block
        dma_cb_seq->info = DMA_NO_WIDE_BURSTS | DMA_WAIT_RESP | \
           DMA_DREQ | DMA_PER_MAP(5);
        dma_cb_seq->src = 0xABCDEF; // Random data
        dma_cb_seq->dst = PWMFIF1_BUS_ADDR;
        dma_cb_seq->length = 4;
        dma_cb_seq->stride = 0;
            // Link to beginning of the sequence if last iteration:
            if (i == (n_cb - 1)) {
                // Link to beginning:
                dma_cb_seq->next = cb_base_bus_addr;
            // Continue:
            } else {
                // Link to next CB:
                dma_cb_seq->next = uncached_virt_to_bus_addr( \
                    cb_base[cb_buf], (dma_cb_seq + 1));
            }
        }

    // Increment control block:
    dma_cb_seq++;
}
```

Configuring and Starting DMA

At this point, the PWM controller and clock manager have been initialized and the control sequence built. The last step is to configure the DMA controller and start the engine to produce the PWM signal. Similar to the PWM controller and CM, a delay between poking registers of 10 microseconds is required for some operations but recommended for all.

Step	Device	Register	Value	Purpose
1	DMA Controller	CS	|= (1 << 30)	Abort current transfer
Delay				
2	DMA Controller	CS	&= ~(1 << 0)	Pause DMA
Delay				
3	DMA Controller	CS	|= (1 << 1)	Clear transfer complete flag
Delay				
4	DMA Controller	CS	|= (1 << 31)	Reset channel
Delay				
5	DMA Controller	CONBLK_AD	Bus address of first CB	Load first CB
Delay				
6	DMA Controller	CS	(7 << 20) | (7 << 16) | (1 << 28)	Set priority. Wait for writes
Delay				
7	DMA Controller	CS	|= (1 << 0)	Start the engine

Setting DMA controller priority and waiting for writes describes its transaction behavior. The transaction priority (0 to 15) refers to what priority any memory transfers have over the peripheral bus. We set this priority to 7 to be kind to any other channels that require a higher priority. Waiting for (outstanding) writes means that the DMA controller will wait for the last outstanding write response to be received before completing the transaction.

Halting DMA

Halting the DMA controller is as important as starting it because the channel persists beyond our program.

Step	Device	Register	Value	Purpose
1	DMA Controller	CS	|= (1 << 30)	Abort current transfer
Delay				
2	DMA Controller	CS	&= ~(1 << 0)	Pause DMA
Delay				
3	DMA Controller	CS	|= (1 << 1)	Clear transfer complete flag
Delay				
4	DMA Controller	CS	|= (1 << 31)	Reset channel