**Objectives**

The objectives of the Direct Memory Access (DMA) is make efficient and effective data transfer between block-to-block memory or peripheral and memory without using CPU. Therefore, CPU can handle others task while the data transfer is handle separately by the DMA. Moreover, the speed in memory transfer of DMA is faster than CPU.

**Hardware Platform**

The tutorial series is written for Raspberry Pi 3/4 B+ board. The main difference is physical peripheral.

base address, which is 0x3F000000 in Raspberry 3, and 0xFE000000 in Raspberry 4.

**Software Requirements**

Make sure that you have the following tools working properly and a driver for mailbox working well.

Visual Studio Code editor

GnuWin32 make tool

aarch64-none-elf compiler

QEMU emulation tool

**Following are the main documents for datasheet and technical references:**

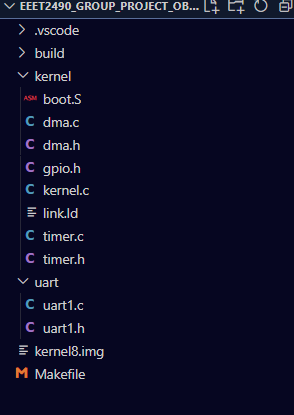
BCM2837 ARM Peripherals - Pi 3

BCM2711 ARM Peripherals - Pi 4

ARM Programmer Guide (for ARMv8-A)

**1. Create the project**

Withthin the project create dma.c, dma.h, timer.c, timer.h



**Add content to dma.c file:**

#include "dma.h"

#include "../uart/uart1.h"

#include "timer.h"

#define SIZE\_LARGE\_DATA 20000000

static dma\_channel \*dma;

dma\_channel channels[15];

static *unsigned* *int* channel\_map = 0x1F35;

// Let the array as global to ultilize global memory region to avoid memory violation

*unsigned* *int* src\_data[SIZE\_LARGE\_DATA];

*unsigned* *int* dest\_data\_CPU[SIZE\_LARGE\_DATA];

*unsigned* *int* dest\_data\_DMA[SIZE\_LARGE\_DATA];

static *unsigned* *int* allocate\_channel(*unsigned* *int* *channel*) {

    if (!(*channel* & ~0x0F)) {

        if (channel\_map & (1 << *channel*)) {

            channel\_map &= ~(1 << *channel*);

            return *channel*;

        }

        return -1;

    }

*unsigned* *int* i = *channel* == CT\_NORMAL ? 6 : 12;

    for (; i >= 0; i--) {

        if (channel\_map & (1 << i)) {

            channel\_map &= ~(1 << i);

            return i;

        }

    }

    return CT\_NONE;

}

dma\_channel \*dma\_open\_channel(*unsigned* *int* *channel*) {

*unsigned* *int* \_channel = allocate\_channel(*channel*);

    if (\_channel == CT\_NONE) {

        uart1\_puts("INVALID CHANNEL!");

        return 0;

    }

    dma\_channel \*dma = (dma\_channel \*)&channels[\_channel];

    dma->channel = \_channel;

    static dma\_control\_block \_\_attribute\_\_((aligned(32))) dma\_block;

    dma->block = &dma\_block;

    dma->block->res[0] = 0;

    dma->block->res[1] = 0;

    REGS\_DMA\_ENABLE |= (1 << dma->channel);

    REGS\_DMA(dma->channel)->control |= CS\_RESET;

    while(REGS\_DMA(dma->channel)->control & CS\_RESET) ;

    uart1\_puts("DMA open successfully!!!");

    return dma;

}

*void* dma\_close\_channel(dma\_channel \**channel*) {

    channel\_map |= (1 << *channel*->channel);

}

*void* dma\_setup\_mem\_copy(dma\_channel \**channel*, *void* \**dest*, const *void* \**src*, *unsigned* *int* *length*, *unsigned* *int* *burst\_length*) {

    /\* Current Transfer Info (TI) setting: 128-bit source read width,

                                            Source address increments after each read. The address will increment by 4, if SRC\_WIDTH=0 else by 32

                                            Use 128-bit destination write width

                                            Destination address increments after each write. The address will increment by 4, if DEST\_WIDTH=0 else by 32

    \*/

*channel*->block->transfer\_info = (*burst\_length* << TI\_BURST\_LENGTH\_SHIFT)

                            | TI\_SRC\_WIDTH

                            | TI\_SRC\_INC

                            | TI\_DEST\_WIDTH

                            | TI\_DEST\_INC;

*channel*->block->src\_addr = (*unsigned* *long*)*src*;

*channel*->block->dest\_addr = (*unsigned* *long*)*dest*;

*channel*->block->transfer\_length = *length*;

*channel*->block->mode\_2d\_stride = 0;

*channel*->block->next\_block\_addr = 0;

}

*void* dma\_start(dma\_channel \**channel*) {

    /\* Convert ARM address to bus address (set higher address bits)

     \* DMA control blocks are located in RAM memory.

     \* Software accessing RAM directly uses physical addresses

     \* (based at 0x00000000). To ensure proper access by the DMA,

     \* the ARM address must be converted to a bus address.

     \* This is achieved by clearing the upper address bits and

     \* setting the base address to 0xC0000000.

    \*/

    REGS\_DMA(*channel*->channel)->control\_block\_addr = (((*unsigned* *long*)(*channel*->block) & (0x3FFFFFFF)) | 0xC0000000);

    REGS\_DMA(*channel*->channel)->control = CS\_WAIT\_FOR\_OUTSTANDING\_WRITES

                          | (DEFAULT\_PANIC\_PRIORITY << CS\_PANIC\_PRIORITY\_SHIFT)

                          | (DEFAULT\_PRIORITY << CS\_PRIORITY\_SHIFT)

                          | CS\_ACTIVE;

}

// Wait until DMA transfer is complete

*int* dma\_wait(dma\_channel \**channel*) {

    while(REGS\_DMA(*channel*->channel)->control & CS\_ACTIVE) ;

*channel*->status = REGS\_DMA(*channel*->channel)->control & CS\_ERROR ? 0 : 1;

    return *channel*->status;

}

*void* dma\_init(){

    dma = dma\_open\_channel(5);

    uart1\_puts("DMA CHANNEL: ");

    uart1\_dec(dma->channel);

    uart1\_puts("\n");

    test\_dma();

}

// Call DMA to start memory transfer with brust\_length = 16 word

*void* do\_dma(*void* \**dest*,const *void* \**src*, *unsigned* *int* *total*) {

*unsigned* *int* ms\_start = timer\_get\_ticks() / 1000;

        dma\_setup\_mem\_copy(dma, *dest*, *src*, *total*, 15);

        dma\_start(dma);

        dma\_wait(dma);

*unsigned* *int* ms\_end = timer\_get\_ticks() / 1000;

    uart1\_puts("The ms take to copy using DMA: ");

    uart1\_dec((ms\_end - ms\_start));

    uart1\_puts("\n");

}

// Test DMA functionality

*int* compare\_memory(const *unsigned* *int* \**a*, const *unsigned* *int* \**b*, *unsigned* *int* *size*) {

    // Compare each value of 2 array

    for (*unsigned* *int* i = 0; i < *size*; i++) {

        uart1\_dec(*a*[i]);

        uart1\_puts("-");

        uart1\_dec(*b*[i]);

        uart1\_puts(" ");

        if (*a*[i] != *b*[i]) {

            return 0; // Memory differs

        }

    }

    return 1; // Memory is the same

}

// Test DMA functionality

*void* test\_dma() {

    uart1\_puts("Testing DMA...\n");

    const *unsigned* *int* src\_data[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

*unsigned* *int* dest\_data[10]; // Initialize destination array with zeros

    // Perform DMA operation (assuming size is in bytes)

    do\_dma(dest\_data, src\_data, sizeof(src\_data));

    uart1\_puts("After do\_dma\n");

    // Verify the copy

    if (compare\_memory(src\_data, dest\_data, sizeof(src\_data)/sizeof(*unsigned* *int*))) {

        uart1\_puts("\nDMA Test PassedVVVV.\n");

    } else {

        uart1\_puts("\nDMA Test Failed.\n");

    }

}

// Compare speed of copy large data set between CPU and DMA

*void* compare\_CPU\_DMA(){

    uart1\_puts("Compare speed between CPU and DMA \n");

    // To save time value

*unsigned* *int* ms\_start;

*unsigned* *int* ms\_end;

    // Initialize source array

    for(*unsigned* *int* i = 0; i < SIZE\_LARGE\_DATA; i++){

        src\_data[i] = i;

    }

    ms\_start = timer\_get\_ticks() / 1000;

    // Copy using CPU

    for(*unsigned* *int* i = 0; i < SIZE\_LARGE\_DATA; i++){

        dest\_data\_CPU[i] = src\_data[i];

    }

    ms\_end = timer\_get\_ticks() / 1000;

    uart1\_puts("The ms take to copy using CPU: ");

    uart1\_dec((ms\_end - ms\_start));

    uart1\_puts("\n");

    // Copy using DMA

    do\_dma(dest\_data\_DMA, src\_data, SIZE\_LARGE\_DATA \* sizeof(*unsigned* *int*));

}

**In dma.h add this content:**

#ifndef DMA\_H

#define DMA\_H

#include "gpio.h"

typedef *struct* {

*unsigned* *int* transfer\_info;

*unsigned* *int* src\_addr;

*unsigned* *int* dest\_addr;

*unsigned* *int* transfer\_length;

*unsigned* *int* mode\_2d\_stride;

*unsigned* *int* next\_block\_addr;

*unsigned* *int* res[2];

} dma\_control\_block;

typedef *struct* {

*unsigned* *int* control;

*unsigned* *int* control\_block\_addr;

    dma\_control\_block block;

} dma\_channel\_regs;

#define REGS\_DMA(*channel*) ((volatile dma\_channel\_regs \*)(*unsigned* *long*)(MMIO\_BASE + 0x00007000 + (channel \* 0x100)))

#define REGS\_DMA\_INT\_STATUS \*((volatile *unsigned* *int* \*)(MMIO\_BASE + 0x00007FE0))

#define REGS\_DMA\_ENABLE \*((volatile *unsigned* *int* \*)(MMIO\_BASE + 0x00007FF0))

//defines for differnet bits of the control and transfer info

#define CS\_RESET            (1 << 31)

#define CS\_ABORT            (1 << 30)

#define CS\_WAIT\_FOR\_OUTSTANDING\_WRITES  (1 << 28)

#define CS\_PANIC\_PRIORITY\_SHIFT     20

    #define DEFAULT\_PANIC\_PRIORITY      15

#define CS\_PRIORITY\_SHIFT       16

    #define DEFAULT\_PRIORITY        1

#define CS\_ERROR            (1 << 8)

#define CS\_INT              (1 << 2)

#define CS\_END              (1 << 1)

#define CS\_ACTIVE           (1 << 0)

#define TI\_PERMAP\_SHIFT         16

#define TI\_BURST\_LENGTH\_SHIFT       12

#define DEFAULT\_BURST\_LENGTH        0

#define TI\_SRC\_IGNORE           (1 << 11)

#define TI\_SRC\_DREQ         (1 << 10)

#define TI\_SRC\_WIDTH            (1 << 9)

#define TI\_SRC\_INC          (1 << 8)

#define TI\_DEST\_DREQ            (1 << 6)

#define TI\_DEST\_WIDTH           (1 << 5)

#define TI\_DEST\_INC         (1 << 4)

#define TI\_WAIT\_RESP            (1 << 3)

#define TI\_TDMODE           (1 << 1)

#define TI\_INTEN            (1 << 0)

typedef *struct* {

*unsigned* *int* channel;

    dma\_control\_block \*block;

*int* status;

} dma\_channel;

typedef *enum* {

    CT\_NONE = -1,

    CT\_NORMAL = 0x81

} dma\_channel\_type;

dma\_channel \*dma\_open\_channel(*unsigned* *int* *channel*);

*void* dma\_close\_channel(dma\_channel \**channel*);

*void* dma\_setup\_mem\_copy(dma\_channel \**channel*, *void* \**dest*, const *void* \**src*, *unsigned* *int* *length*, *unsigned* *int* *burst\_length*);

*void* dma\_start(dma\_channel \**channel*);

*int* dma\_wait(dma\_channel \**channel*);

*void* dma\_init();

*void* do\_dma(*void* \**dest*,const *void* \**src*, *unsigned* *int* *total*);

*void* test\_dma();

*void* compare\_CPU\_DMA();

#endif

**In timer.c add this parameter:**

#include "timer.h"

// Timer and counter to get current ticks

*void* timer\_init() {

*unsigned* *int* cur\_val\_1 = 0;

    cur\_val\_1 = REGS\_TIMER->counter\_lo;

    cur\_val\_1 += 1000000;

    REGS\_TIMER->compare[1] = cur\_val\_1;

}

*unsigned* *long* timer\_get\_ticks() {

*unsigned* *int* hi = REGS\_TIMER->counter\_hi;

*unsigned* *int* lo = REGS\_TIMER->counter\_lo;

    //double check hi value didn't change after setting it...

    if (hi != REGS\_TIMER->counter\_hi) {

        hi = REGS\_TIMER->counter\_hi;

        lo = REGS\_TIMER->counter\_lo;

    }

    return ((*unsigned* *long*)hi << 32) | lo;

}

**In timer.h add these line:**

#include "gpio.h"

// 10.2

*struct* timer\_regs

{

*unsigned* *int* control\_status;

*unsigned* *int* counter\_lo;

*unsigned* *int* counter\_hi;

*unsigned* *int* compare[4];

};

#define REGS\_TIMER ((volatile *struct* timer\_regs \*)(*unsigned* *long*)(MMIO\_BASE + 0x00003000))

*void* timer\_init();

*unsigned* *long* timer\_get\_ticks();

**Modify the kernel.c:**

#include "../uart/uart1.h"

#include "dma.h"

*void* main()

{

    // set up serial console

    uart1\_init();

    // say hello

    uart1\_puts("Hello World!\n");

    // set up DMA

    dma\_init();

    // See how fast CPU vs DMA in copying memory

    compare\_CPU\_DMA();

    // echo everything back

    while(1) {

        //read each char

*char* c = uart1\_getc();

        //send back

        uart1\_sendc(c);

    }

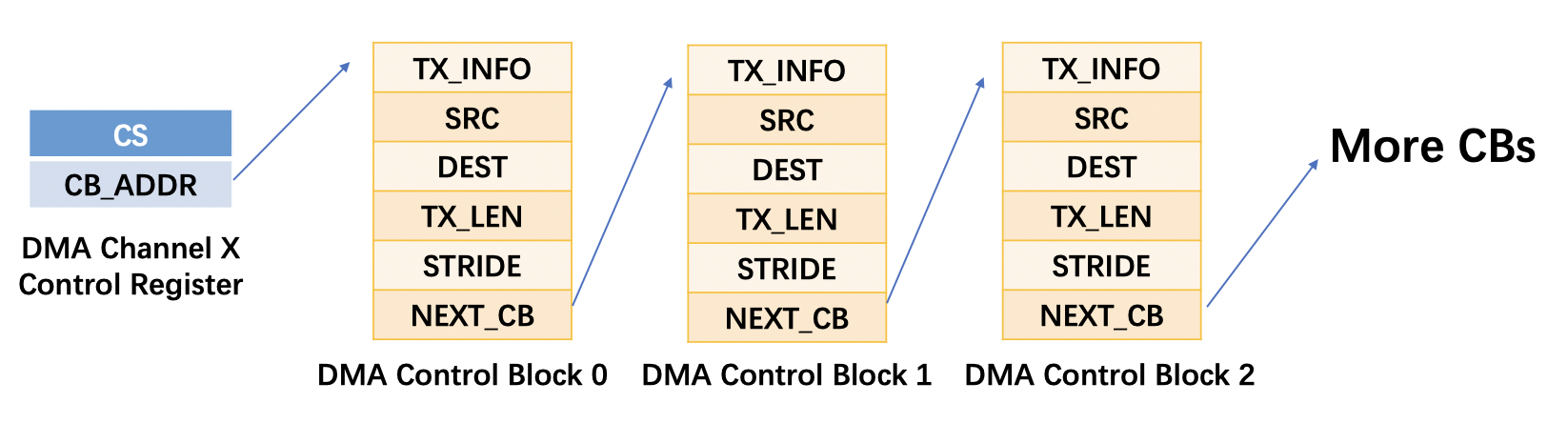
}

**Explaination:**

[fandahao17/Raspberry-Pi-DMA-Tutorial: A step-by-step guide on using DMA to achieve high-speed sampling on Raspberry Pi (github.com)](https://github.com/fandahao17/Raspberry-Pi-DMA-Tutorial)

Both Pi3 and Pi4 have 16 DMA channel and each of them have a Control Block and they can setup as a linked list therefore to make a continuous memory copy function. When DMA starts working, the cb\_addr field is updated after each DMA transfer so it always point to the current control block. The difference between the Pi4 and Pi3 is that Pi4 have Four of these are DMA Lite channels (with reduced performance and features)(Pi3 also have Lite engine but is not carefully describe in the document), and four of them are DMA4 channels (with increased performance and a wider address range). The DMA Lite in Pi4 channel are: 7, 8, 9, 10 and DMA4 are: 11, 12, 13, 14, others are normal DMA. There is exception that DMA Channel 15 is physically removed from the other DMA Channels and so has a different address base of 0x7ee05000. DMA Channel 15 is exclusively used by the VPU.

To start a transfer process of DMA, write the address of a Control Block (CB to the CONBLK\_AD register and set the ACTIVE bit. The DMA will read the CB from this address, loads it into read-only registers, and begins the data transfer. When the transfer completes, the DMA updates the CONBLK\_AD register with the NEXT\_CB register's value, fetches a new CB, and repeats the process. To make DMA stops set NEXT\_CB is 0x00000000 and clearing the ACTIVE bit. Most CB registers are read-only and updated automatically, but NEXT\_CB 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.



The dma\_init() function is to initalize the DMA Controller of the chip which the channel 5 to be specific. It is start with allocate a valid channel number and check if whether the channel is valid or not. After that it aligned a dma\_block aka a Control Block of DMA to a 32-bit boundary and call register to enable DMA. Since then call the test\_dma() to verify the DMA functionality.

To make start a memory transfer write source and destination bus address and the total of byte need to be transfter to the do\_dma() function with default brust\_length of 16 words.  
  
**Build and Run**

To run the project in type: “make” it will compile and run the emulator. If you want to compile only type: “make compile” to only make a .img file. Attribute “pi=4” can be add to specific board Pi4 to compile like “make pi=4”.

Build the whole project, and run it on QEMU. You will see the information printed on the console and on QEMU window (Raspberry Pi’s emulated screen) as below:  
A computer screen shot of a code

Description automatically generated

Note: This is QEMU 9.0 and it can emulate Pi4.  
The interesting part that using DMA on QEMU emulator is very slow compare to CPU copy.

However, things change when we use the real Pi4 board, this is the Tera Term connect UART to Pi4 using UART1:

Note: Remember to adjust definition of MMIO\_BASE to 0xFE000000, before building the project again for RPI4.

A screenshot of a computer program

Description automatically generated  
  
As we see the speed to copy 20.000.000 unsigned integer using DMA is 66 times faster than using CPU with only 103ms!!!  
  
Note: The submit zip file have Makefile to run with QEMU 3.x.x