# DMA

By Brandon Whitehead.

The best way to think about DMA is that it is a super fast for loop thats done in hardware.

Take this snippet of C code for copying an array into another array

```
u16 heykiddies[20] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
16, 17, 18, 19};
u16 thosekiddies[20];

for (int i = 0; i < 20; i++)
     thosekiddies[i] = heykiddies[i];
```

Now this code is obviously copying something from a source to a destination. Now you can rewrite this as a function to handle copying any two pointers from source to destination

```
u16 heykiddies[20] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
16, 17, 18, 19};
u16 thosekiddies[20];
heycopyatob(heykiddies, thosekiddies, 20);

void heycopyatob(u16* src, u16* dst, int size)
{
    while(size)
    {
        // Copy element from source to destination.
        *dst = *src;
        src++;
        dst++;
        size--;
    }
}
```

So this is really what DMA does, copy things. The DMA registers are highly configurable allowing you to specify how to copy. So you can tell it to increment the source or destination, decrement the source or destination, or do nothing with the source or destination.

**You need 4 things to be able to use DMA effectively.**

1. What is my source address?
2. What is my destination address?
3. How many elements do I need to copy?
4. How do I want the copy to be performed?

Here are all of the flags for DMA (I have included this as an attachment called DMA.h)

**This set of flags will modify the line dst++ in my code above.**
#define DMA_DESTINATION_INCREMENT (0 << 21)
#define DMA_DESTINATION_DECREMENT (1 << 21)
#define DMA_DESTINATION_FIXED (2 << 21)
#define DMA_DESTINATION_RESET (3 << 21)
Remember that 0 left shited by anything is zero, therefore it is the default.

1. If I said Decremenet instead then I would do dst--
2. If I said Fixed then remove that line of code,
3. Reset is the least useful of these forget about it (if you really want to know at the end of the dma it will reset the destination back to where it was when the DMA started.)

**This set of flags will modify the line src++**
#define DMA_SOURCE_INCREMENT (0 << 23)
#define DMA_SOURCE_DECREMENT (1 << 23)
#define DMA_SOURCE_FIXED (2 << 23)

1. If I said Decremenet instead then I would do src--
2. If I said Fixed then remove that line of code,

This set of flags will modify the parameters to the above function u16* src, u16* dest.
#define DMA_16 (0 << 26)
#define DMA_32 (1 << 26)

1. If DMA_32 is given then change the paramters to u32* src, u32* dst we will now copy 32 bits at a time. DMA_16 is the default which is most useful for mode 3.

**These are the DMA timing options not very useful but they are there if you need it**
#define DMA_NOW (0 << 28)
#define DMA_AT_VBLANK (1 << 28)
#define DMA_AT_HBLANK (2 << 28)
#define DMA_AT_REFRESH (3 << 28)

1. You can start the DMA immediately (given DMA_ON is given as well)
2. You can start the DMA at a vertical blank
3. You can start the DMA at horizontal blank
4. The last is not useful and I have not used it before don't worry what it does.

**This flag is used in conjuction with the above to repeat everytime it is necessary**
#define DMA_REPEAT (1 << 25)

**This flag will allow you to set a interrupt handler function to be done when the DMA is**

**finished**
#define DMA_IRQ (1 << 30)
You can read tonc about interrupts on the GBA to be able to use this one.

**Master swtich which is a kick for DMA to start**
#define DMA_ON (1 << 31)

So going back to my example where I wanted to copy an array to another array

```
u16 heykiddies[20] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
16, 17, 18, 19};
u16 thosekiddies[20];
heycopyatob(heykiddies, thosekiddies, 20);

void heycopyatob(u16* src, u16* dst, int size)
{
    while(size)
    {
        // Copy element from source to destination.
        *dst = *src;
        src++;
        dst++;
        size--;
    }
}
```

One can rewrite this using DMA as follows.

```
u16 heykiddies[20] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
16, 17, 18, 19};
u16 thosekiddies[20];
DMA[3].src = heykiddies;
DMA[3].dst = thosekiddies;
DMA[3].cnt = 20 | DMA_SOURCE_INCREMENT | DMA_DESTINATION_INCREMENT | DMA_16 |
DMA_NOW | DMA_ON;

Remember that most of these flags are the defaults.  I personally don't
prefer to type all of those things but hey more power to you if you do.

DMA[3].src = heykiddies;
DMA[3].dst = thosekiddies;
// Default is to increment source, increment destination, copy 16 bits at a
time, and to do the DMA now.
DMA[3].cnt = 20 | DMA_ON;
```

Note there are multiple ways to access the DMA registers as per the header file you can use this set of declarations to be able to access them.

/* DMA channel 3 register definitions */
#define REG_DMA3SAD *(volatile unsigned int*)0x40000D4 /* source address*/
#define REG_DMA3DAD *(volatile unsigned int*)0x40000D8 /* destination address*/
#define REG_DMA3CNT *(volatile unsigned int*)0x40000DC /* control register*/

The above would look like this using this set of declarations.
REG_DMA3SAD = (volatile unsigned int*) heykiddles;
REG_DMA3DAD = (volatile unsigned int*) thosekiddies;
REG_DMA3CNT = 20 | DMA_ON;

Now if you are going whoa whoa whoa whats with the casts? then these declarations were written to accept a variable of type volatile unsigned int* so when assigning these pointers you must ensure that they point to the same type else you will get a compiler error.

## Second Example

So take this code for filling an array (that is copying the same value) to each location in this array.

```
short x = 32;
short fillthiswithx[256];

for (int i = 0; i < 256; i++)
    fillthiswithx[i] = x;
```

Now answer the four questions.

**What is my source address?** This one is tricky. Remember that I asked SOURCE ADDRESS so your answer needs to be of a pointer type. If I just said x, x is not a pointer it is a short. However saying &x is a pointer of type short*. Therefore the source address is &x. If you do not understand this see the function heycopyatob again.
**What is my destionation address?** fillthiswithx
**How many elements do I need to copy?** 256
**What control flags do I need?** The source does not change so I do not want to increment the source address.

So with these four answers I can now change that into a DMA call.

DMA[3].src = &x;
DMA[3].dst = fillthiswithx;
DMA[3].cnt = 256 | DMA_SOURCE_FIXED | DMA_ON;

Also note that your DMA source may need to be volatile. The optimizer may sometimes play tricks on your code if the optimizer sees an opportunity to make your code more efficient. So x in the above example should be made volatile.

So here is my suggestion.  Even though that these are toy examples.  Write it out on the gameboy.  You also do know that I have gotten printf to work on the gameboy advance.  Try out the code I have given to you above and make sure it works by printing out each element in those arrays.  Play around with it with the various DMA flags and reason why did that happen.

The reason why this is a useful thing to learn is that this is an application of pointers and pointer arithmetic.  In my example above you had to remember to always give DMA an address.  Well how do you get the address of a variable?  you'd use the address of operator to get that.

This ends my note on DMA.  You should be able to extend this and be able to draw a rectangle onto the screen using DMA (as it was done in class).  You will be given problems on DMA in lab on Monday.  So I really hope you all read this before then!

but the big thing to remember from this is to write it out using a for loop.  DMA will effectively become a for loop, just a for loop done in hardware.