DMA: A Crucial Resource in Embedded and General Purpose O.S. Design

Anthony Redamonti *(Author)*
Dept. of Computer Science
Syracuse University College of Engineering & Computer Science
Syracuse, NY, USA
aeredamo@syr.edu

***Abstract*—Background: Most computing applications require some transfer of data, which can range from a single bit to terabytes in size. For the CPU to handle data transfer, it would need to perform programmed I/O, which consists of watching status bits and feeding data into a control register on a per byte basis. A side effect of the CPU handling the transfer of data would be long delays in the execution of applications. High-frequency CPU's can improve performance but are costly and consume more power than standard CPU's. Direct Memory Access (DMA) relieves the CPU from performing the arduous task of data transfer by allowing hardware sub-systems (peripheral devices) to access main memory (RAM) independent of the CPU.**

Keywords: DMA, DMAC, CPU, RAM, Flash, Interrupt, Buffer, Real-time Operating System (RTOS), Peripheral Device, Peripheral Component Interconnection (PCI) Bus, Memory Mapping, Cache, Digital to Analog Convertor (DAC), Analog to Digitial Convertor (ADC)

I. Introduction

When transferring blocks of data between disk and RAM, there are two options. The first is to have the CPU read each byte from a disk controller and store the contents in the appropriate memory location. Each byte would require the CPU to perform a series of fetch-decode-execute instructions, inhibiting the execution of other tasks. The second is to use DMA to perform the data transfer independent of the CPU.

To illustrate the time saved using DMA, consider the following scenario when the CPU is used to perform a data transfer of 10 bytes from Flash to SRAM. The following loop must be performed once per byte (10 times):

1. Copy the value in Flash and store it in a general-purpose register.
2. Copy the value in the general-purpose register to the Accumulator register.
3. Copy the Accumulator value to the desired destination memory location in SRAM.
4. If all data have been transferred, end. If not, increment the Flash pointer.
5. Increment SRAM pointer.
6. Jump to step 1.

The timing diagram below (Fig. 1) represents the same data transfer using DMA [1]. The first few cycles represent DMA initialization. After initialization, each byte consumes one clock cycle with the hardware handling pointer increments in between each cycle. After the last byte is transferred and the end of the transfer is verified, a signal is sent to notify the CPU. Because the workload of the CPU is diminished, less power is consumed. The data transfer is significantly faster using DMA as the DMA hardware performs logic at a byte-per-cycle rate.

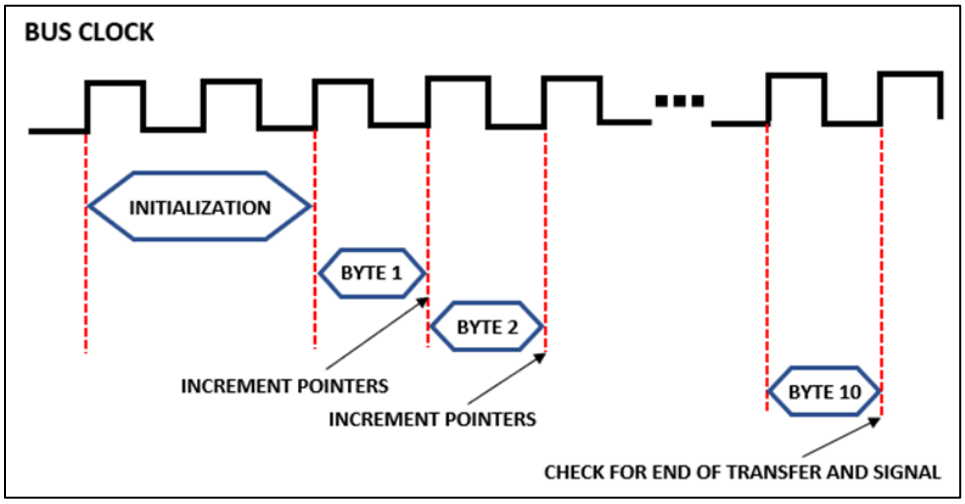

Fig. 1. Timing Diagram of DMA Data Transfer

II. Standard DMA Modes

DMA can handle both high and low speed applications. It is ideal for high speed transfers like USB or Ethernet where packets are being transferred in blocks between memory and I/O. Because DMA does not experience the same latency as the CPU, it is perfect for maintaining packet timing, which is crucial for most high-speed communication protocols. For low speed applications, DMA can store data in its buffer as it is passed from the source and interrupt the CPU to gain access to the memory bus if needed.

There are typically three modes of DMA: block mode, burst mode, and transparent mode [2].

A. Block Mode

In *block mode*, the CPU is not allowed access to the memory bus until the transfer of the data block has completed. Block mode is ideal for situations where the transfer is of the highest priority, such as a page fault. However, when in block mode, CPU starvation is correlated with the size of the data block. Block mode is initiated and controlled via software.

B. Burst Mode

The second type of DMA is *burst mode*, also known as cycle stealing mode. In burst mode, the DMAC will periodically requests control of the memory bus for a few cycles before releasing it back to the CPU. The process continues until the transfer is finished. Burst mode is the most popular type of DMA as it represents a compromise between optimal CPU performance and fast DMA transfer speeds.

C. Transparent Mode

The third type of DMA is *transparent mode* which only uses the memory bus when it is not in use by the CPU. One can see how the three modes are all viable options that a programmer can choose depending on the importance of the data being transferred versus the importance of the processes being executed by the CPU.

III. Additional DMA Modes

The DMA Controller (DMAC) is a special-purpose processor which performs high-speed data transfer between an I/O device and the memory. Thus, the CPU and DMAC must share the system bus. The most common approach to bus sharing is through a special control register on the DMAC which requests control of the bus from the CPU. The DMA device usually includes a temporary storage register used to buffer data between load/store commands.

There are three additional modes of DMA: *Demand Mode*, *Fly-by Mode*, and *Buffer Chaining Mode* [3].

A. Demand Mode

Demand Mode allows an I/O device to control and synchronize the movement of data between itself and memory.

B. Fly-by Mode

Fly-by Mode allows data transmission between two devices in one access cycle by performing read/write operations at predetermined locations in each device. Because the locations are predetermined, the operation bypasses the temporary storage register, allowing faster data transmission (one access cycle). However, memory-to-memory transfers are not possible in this mode.

C. Buffer Chaining Mode

Buffer Chaining Mode utilizes a linked list in memory to store the data. A descriptor at the beginning of the linked list provides the byte count, destination address, source address, and a pointer to the next memory descriptor. The transfer will continue until the number of bytes transferred equals the number of bytes in the byte count field.

Not all DMAC devices support all types of DMA transfer. Some of them perform data transfer by performing a similar routine as the CPU through reading data from the source address, storing it in the temporary storage register, and loading it to the destination address. However, unlike the CPU, DMA controllers do not have to perform the "fetch-decode-execute" cycle as they do not have to execute code. Therefore, the rate of data transfer achieved through DMA controllers will always be superior to that of the CPU.

IV. Dynamic Scheduling Using DMA

Srikanth. K discusses the use of DMA when performing dynamic rescheduling in real-time operating systems (RTOS) in embedded applications [4].

The RTOS allows the user to change the priority of tasks during runtime through dynamic scheduling. DMA is used to handle dynamic priority scheduling input from the user, freeing up the CPU to perform more critical tasks. By preventing the CPU from idling for long periods of time, the system reliability and performance is improved.

The Samsung S3C2440 ARM9 microprocessor supports four DMA controller registers located between the system bus and the peripheral bus. Because the data transfer source and destination addresses are programmable, each channel can manage bidirectional data transfer between the peripheral bus and the system bus. The DMA interacts with the user by collecting time and date information stored in the EEPROM by the real-time clock (RTC) and sending it to the LCD display via the SPI Bus.

While the DMA performs these dynamic tasks at runtime, the CPU services other tasks. Each DMA channel has a priority and its own interrupt used to trigger an event. When the DMA has finished transferring data to the LCD display, the CPU is notified via interrupt. While DMA can handle asynchronous user input via external interrupt, it can also be programmed to handle other synchronization (SYNC) events such as synchronous events via peripheral timer registers.

V. THE DMA CONTROLLER (DMAC)

A.F. Harvey details the responsibility of the DMA Controller (DMAC) during each step of a transfer from a peripheral device to internal memory [5].

In *Basic Transfer Mode*, the device requesting DMA must first send the *bus request* signal to the CPU. After completing the current bus cycle, the CPU will send the *bus grant* signal shown in Figure 2 below. The device will send the *bus grant ack* (acknowledge) signal, and the CPU will begin monitoring the bus.

The DMAC saves the programmed address and count in the base registers and copies the information in the current address and current count registers shown in Figure 1 above. The DMA mask register is used to enable or disable each DMA channel. To initiate the transfer, the base registers (Base Count and Base Address) are written and copied to the current registers (Current Count and Current Address), and the DMA channel is enabled. With each DMA transfer, the Current Address Register is transmitted on the bus and incremented or decremented thereafter. During data transmission, the CPU must check if the DMA is changing the value of addresses who have a copy stored in cache memory. If there is a copy in cache, the CPU must either update the copy or invalidate it.

The Current Count Register represents the number of transfers remaining, and when it transitions from 0 to -1, the *terminal count* (TC) signal is produced, signifying the completion of the DMA transfer. The TC signal is also referred to as a TC pulse, which is often monitored by I/O devices included in the transfer. The CPU must reprogram the DMAC upon reaching the TC event. The time spent reprogramming the DMAC is much less than the time it would take to service an I/O interrupt. The DMA controllers can also reprogram themselves through *autoinitialization*. The process involves reloading the channel's current registers from the base registers and reenabling the channel. The status registers, read by the CPU, convey the state of the DMA channel. Once the transfer has been complete, the device will release the bus by sending the bus release signal. The CPU will then resume normal bus cycles from the point at which it was interrupted.

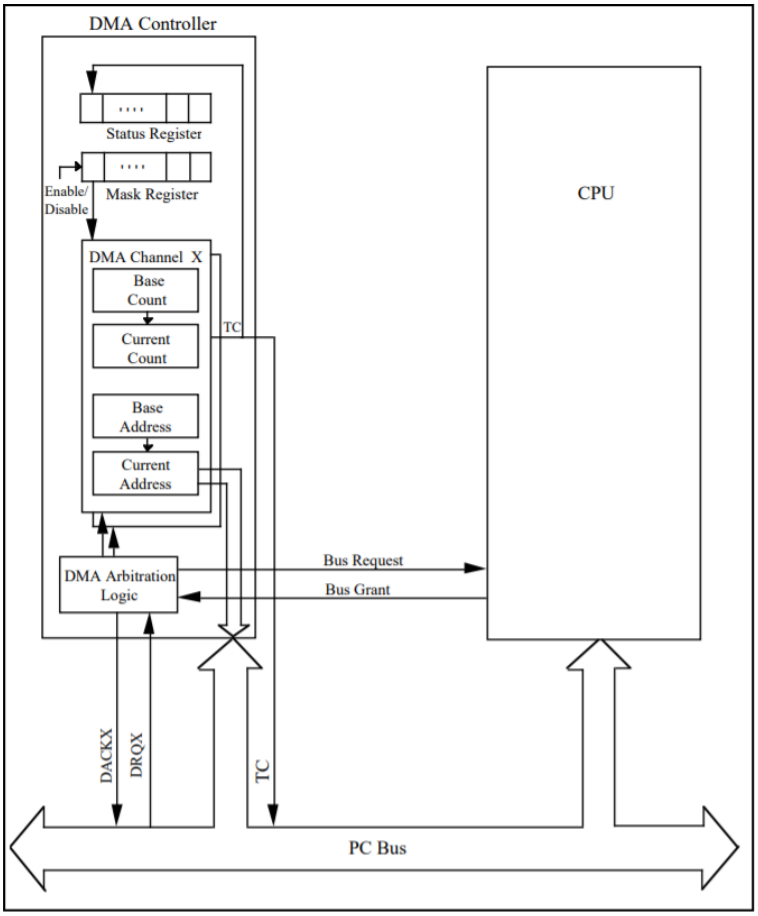

Fig. 2. DMA Controller Architecture

VI. The Functionality of DMA Buffers

Alessandro Rubini outlines how *DMA buffers* are used during transfers between peripheral devices and internal memory [6].

External storage devices such as portable hard disks (HDDs) or solid-state drives (SSDs) are common among personal computers and other machines. These devices have a controller, such as a disk controller, which typically have a DMA buffer to transfer data to and from the DMAC. DMA is invoked through software via function call or by hardware asynchronously supplying data to a DMA buffer to be read later.

The chain of events for a process to invoke the use of DMA are as follows. First, the process calls *read*, which will signal the peripheral device driver method to allocate a DMA buffer and initiate a data transfer. The process is then put to sleep. The peripheral device sends an interrupt signal to the CPU after the transfer has completed and the buffer is empty.

Peripheral devices will supply data to the system asynchronously using DMA. The peripherals can do this without interaction from the CPU by storing data in the DMA buffer. The DMA buffer is maintained by the device driver so that if the user calls *read*, the data will be transferred to the CPU. When the peripheral device is ready to transfer new data, the device driver raises an interrupt to announce the commencement of data transfer. The interrupt handler will allocate a DMA buffer for the device to offload the data. When the device has filled the buffer, it will send another interrupt. The interrupt handler will offload the data from the buffer, emptying it, and will wake any relevant blocked processes waiting for the data.

VII. DMA Mapping

Avi Silberschatz discusses the use of mapping registers to perform DMA Mapping [7].

DMA buffers are allocated and given an address through a process known as *DMA mapping*. The address assigned to the buffer must be accessible to the device and is created by calling *virt_to_bus*(). Some hardware use *mapping registers* which remap the limited range of addresses of the peripheral devices that perform DMA, essentially performing virtual memory for the peripherals. The process is known as *direct virtual memory access* (DVMA), and it allows two mapped devices to perform a DMA transfer without the use of the CPU or RAM.

Other techniques of rearranging peripheral addresses involve the use of bounce buffers. Most peripherals cannot access high memory addresses. If a device driver attempts to perform DMA on an address space that is not accessible by the peripheral device, data can be copied to and from the bounce buffer to access the data.

An operating system with a protected-mode kernel will prevent low-privilege processes from being able to send or receive DMA commands to or from peripheral devices. System reliability and safety is improved by restricting the use of DMA to only high-priority processes. The probability of incorrect DMA use by device drivers would decrease, preventing system crashes. An OS kernel without memory protections provides access to DMA for all processes, yielding higher throughput of data by avoiding context switching with the kernel. It also decreases system safety and reliability, which is why most general-purpose operating systems use a protected-mode kernel.

VIII. Double Buffers

For some applications, it is necessary for the CPU to perform calculations on data while the DMA is transmitting the data to/from a peripheral in real-time. For such scenarios, a *ping-pong* buffer is used [8].

A ping-pong buffer is a pair of buffers used separately by the CPU and DMA. One buffer can be labelled the background buffer, which will be used by the DMA to transmit data between memory and a peripheral device. The other buffer can be labelled the foreground buffer, which is used by the CPU to perform calculations (for example, a Fast-Fourier Transform). When the DMA fills the background buffer, an interrupt is triggered, which will switch the pointers for each buffer. By allowing the background buffer to remain active, data can be processed in real-time by the CPU.

The most common applications involving ping-pong buffers, also called *double buffers*, are streaming media applications (audio, video, etc.). Data on one end is being streamed from a source, like the internet or a network, while being processed by an audio or graphics card. One could imagine the awful user experience of attempting to stream media using a single buffer.

IX. Scatter-Gather Lists

John Franco outlines the use of multi-page scatter-gather lists in DMA applications [9].

Consider a multi-page buffer created in user space. The buffer will have a continuous virtual memory but will represent scattered pockets of available space in physical memory. For a peripheral device to perform DMA on the buffer in a single operation, the device driver must either copy the data into a large, contiguous space in physical memory or use a *scatter/gather list*. A *scatter/gather list (scatterlist)* is a structure consisting of a series of array pointers and their corresponding lengths.

```
struct scatterlist {
        struct page*    page;
        unsigned int    offset;
        dma_addr_t      dma_address;
        unsigned int    length;
};
```

In the scatterlist structure described in <asm/scatterlist.h>, there are four fields. The pointer to the structure page points to the page in memory containing the buffer to be used in the scatter/gather operation. The offset and length represent the length of the buffer and its starting location within the page. The DMA address represents an address of the peripheral device at which DMA is being performed for this page.

When altering pages in a scatterlist, it is imperative to inform the kernel of which pages have been altered. The driver must call "*SetPageDirty(struct page *page);*" before releasing the page. If the kernel is not informed of a dirty page, the changes to that page are not copied into main memory. The driver should also free the dirty pages stored in cache with a call to "*page_cache_release(struct page *page);*".

X. Enhanced DMA (EDMA)

How would DMA manage multiple real-time data transfers? The DMAC must be able to manage real-time asynchronous I/O streams based on their priority. Enhanced DMA (EDMA) uses preemption and pipelining to accomplish this task [10].

Typically, the I/O scheduler will not allow preemption of an I/O request. However, when there are multiple real-time I/O streams in a system being managed by DMA, the DMAC must allow preemption. Preemption is priority based, meaning that if one transfer interrupts another, the higher priority transfer will resume control of the bus. The priority of the peripheral device is assigned by the programmer. If a device is transmitting a real-time signal, it will have a higher priority than a non-real-time signal. If multiple real-time signals are transmitting, the faster signal will have higher priority. Preemption requires that transfer requests be queued according to their priority levels.

The DMAC must reprogram the channel when a transfer is initiated or resumed. Thus, a high rate of preemption could create latency issues due to the DMAC channel reprogramming time as the bus would remain idle during these periods. To prevent this inefficiency, EDMA uses *pipelined command bus protocol*. When one transfer is active, the control information of the next transfer stored in the queue is programmed on another channel waiting to use the bus. EDMA can also perform *linking*, which is based on the principles of spatial and temporal locality. Linking mimics prefetching in that if an address is or is near (adjacent to) an address that has been modified recently, it is prudent to store the address in the DMA registers. Ping-pong and circular buffers are data structures that are utilized more efficiently through linking.

XI. DMA Performance Test

As previously stated, DMA chips have several channels, each with unique priority. There are four priority levels: low, medium, high, and very high. DMA channels can be programmed to use *circular buffers*. A circular buffer, also called a *ring buffer*, is a structure which treats a fixed size array as if the end of the array is connected to the beginning, forming a circle. Once the buffer has been filled, values are overwritten starting with the first index in the array. Thus, the circular buffer seemingly has no limit, making them ideal for buffering data streams [11].

DMA channels work together to limit the use of the memory bus. When one channel is done with a transfer and needs to be reprogrammed by the controller (i.e. increment destination address), another channel is ready to transfer. One important note is that even when DMA is performing a transfer, it is not occupying all the bus cycles. On an Advance High-performance Bus (AHB), DMA occupies 5 cycles, leaving 3 left over for the CPU. Therefore, the CPU can access any peripheral memory address during a DMA transfer because DMA is only using the bus a maximum amount of 40% of the time.

The functions below were used to measure the data transfer rates of the CPU versus the DMA on two blocks of data of equal size (800 uint32_t = 3200 bytes). The data transfers were memory to memory (no peripherals involved).

```
#include "stm32f10x.h"
#include "leds.h"
#define ARRAYSIZE 800
volatile uint32_t status = 0;
volatile uint32_t i;

int main(void)
{
   //initialize source and destination arrays
   uint32_t source[ARRAYSIZE];
   uint32_t destination[ARRAYSIZE];
   //initialize array
   for (i = 0; i < ARRAYSIZE; i++)
      source[i] = i;
   //initialize led
   LEDsInit();

   /* DMA settings here */

   //send values to DMA registers
   DMA_Init(DMA1_Channel1, &amp;
DMA_InitStructure);
   // Enable DMA1 Channel Transfer Complete interrupt
   DMA_ITConfig(DMA1_Channel1, DMA_IT_TC,
ENABLE);

   //LED on before the transfer
```

```
    LEDToggle(LEDG);

    //Enable DMA1 Channel transfer
    DMA_Cmd(DMA1_Channel1, ENABLE);
    while (status == 0) {};

    LEDToggle(LEDB);
    for (i = 0; i < ARRAYSIZE; i++)
    {
      destination[i] = source[i];
    }
    LEDToggle(LEDB);

    while (1)
    {
      //interrupts do the job
    }
}

void DMA1_Channel1_IRQHandler(void)
{
    //Test on DMA1 Channel1 Transfer Complete interrupt
    if (DMA_GetITStatus(DMA1_IT_TC1))
    {
      status = 1;
      LEDToggle(LEDG);
      //Clear DMA1 Channel1 Half Transfer, Transfer
Complete and Global interrupt pending bits
      DMA_ClearITPendingBit(DMA1_IT_GL1);
    }
}
```

The LEDG and LEDB are used to measure the performance of the DMA and CPU respectively. After the DMA settings are configured, the channel and DMA interrupt are initialized. LEDG is toggled before enabling the DMA transfer. An empty while-loop is used to wait for the status variable to be set by the DMA interrupt, signifying completion. The DMA interrupt also toggles the LEDG bit for measuring the elapsed time of the transfer. LEDB is toggled before entering a for-loop where each entry in the array is individually transferred by the CPU. The LEDB is toggled signifying the end of the CPU transfer. The elapsed time of the DMA transfer was 214 µsec, while the elapsed time of the CPU transfer was 544 µsec.

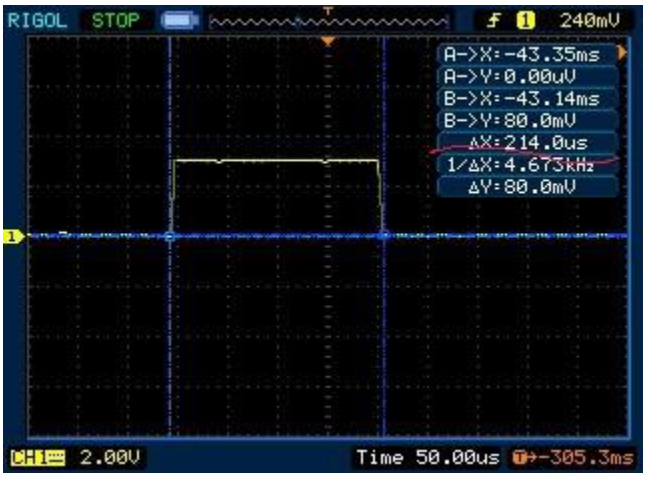

Fig. 3. Scope Trace of LEDG (DMA Transfer = 214.0 µsec)

Fig. 4. Scope Trace of LEDB (CPU Transfer = 544.0 µsec)

XII. Copley Serial Resolver

A. *Background*

It is common in servo-motor applications to use an *encoder* to track changes in motor position. To provide feedback for applications set in harsh weather environments where system reliability is essential, a popular choice is the *resolver*. A resolver is essentially a transformer that determines the angle and velocity of the motor shaft. Because there are no integrated circuits in a resolver, it has a very wide temperature tolerance. It can also withstand high amounts of mechanical vibration and even radiation.

A resolver is comprised of a stationary part called a *stator* and a rotary part called a *rotor*, which is attached to the motor shaft. A sinusoidal voltage signal is used to excite the primary winding of the rotor, which generates two secondary sinusoidal output waveforms from the stator serving as feedback. The secondary waveforms are 90

degrees out of phase so one is called sine and the other cosine. Together they are used to determine the angle and velocity of the motor shaft.

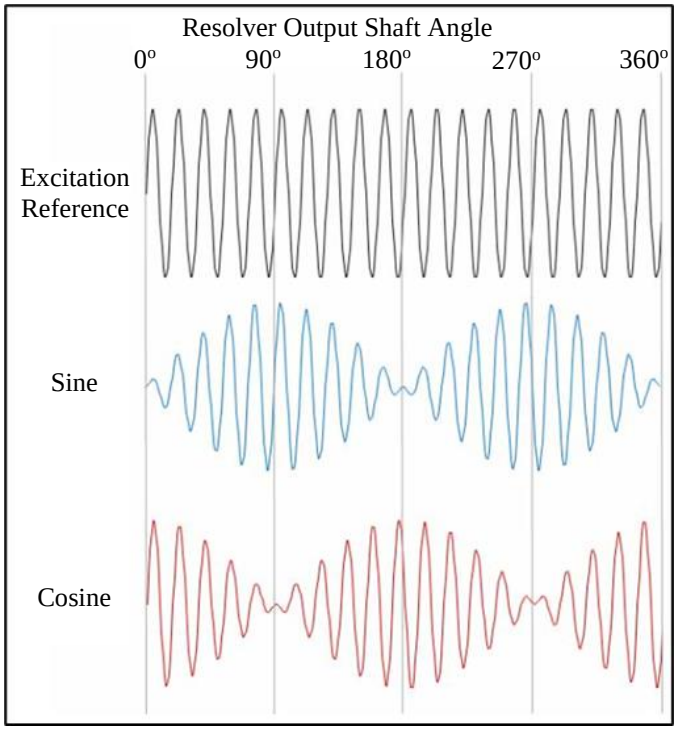

Fig. 5. Graph of Resolver Waveforms [12]

Copley Controls has been an industry leading servo drive manufacturer since 1980. Their products have a wide range of innovative features to provide the best custom solutions in the motion control industry. Most of the Copley product line has two versions of each product. One version supports many feedback types and the other contains special hardware to support only resolver feedback. To avoid manufacturing two versions of every product, Copley engineering has developed the Copley Serial Resolver Board. The board can interface with any existing model servo drive to add resolver feedback capability, making each model's resolver version obsolete.

Within the board is an STM32G0x1 advanced ARM-based 32-bit microcontroller. The sinusoidal output voltage used to excite the resolver is generated by the 12-bit digital-to-analog converter (DAC) module on the MCU. Each DAC channel has DMA capability. The sine wave is generated using a lookup table in memory. Because DMA is used to write values from the lookup table to the DAC data register (DAC_DHR12R1), system reliability is improved. If the CPU was used to perform the transfer, it would use interrupts at a high frequency, creating system latency.

B. Implementation: DAC

The STM32G0x1 Peripheral Register Boundary Address Table in the reference manual contains the address boundaries for each I/O port or bus. Within the table is the address range for the Advanced Peripheral Bus (APB), which defines the address boundaries for all peripheral devices. The address boundary for the DMA registers is defined as 0x40020000-0x400203FF (1KB). The struct "DMAregs," shown below, defines each register in the boundary.

```
#define DMA_BASE              0x40020000
typedef struct
{
        REG intStat;  // DMA interrupt status register
                      // (DMA_ISR)
        REG intClr;   // DMA interrupt flag clear register
                      // (DMA_IFCR)
        struct
        {
                REG config; // DMA channel x
configuration register (DMA_CCRx)
                REG count;  // DMA channel x number
of data to transfer register (DMA_CNDTRx)
                REG pAddr;  // DMA channel x
peripheral address register (DMA_CPARx)
                REG mAddr;  // DMA channel x
memory address register (DMA_CMARx)
                REG rsvd;    // reserved
        } channel[7];
} DMAregs;
```

Below is the code to initialize DMA channel 1 to update the DAC value.

```
// DMA channel 1 is used to update the D/A value every
time the timer updates.
dma->intClr = 0x000000FF;
dma->channel[0].config = 0x00000000;
dma->channel[0].count = TBL_LEN;
dma->channel[0].pAddr = (uint32_t)&dac->r12;
dma->channel[0].mAddr = (uint32_t)sineTbl; dma-
>channel[0].config = 0x000035B6;
```

First, the value 0x000000FF is written to the DMA Interrupt Flag Clear Register, which clears the transfer error flags and global interrupt flags for DMA channels 1 and 2. Next, the channel 1 configuration register is reset, and the number of data to transfer is set to TBL_LEN (100). The peripheral address was set to the DAC channel1 12-bit right-aligned data holding register, and the memory address was set to the beginning of the lookup table (sineTbl).

The channel 1 configuration register is set to 0x000035B6. Bits 12 and 13 represent the priority level of the DMA channel and are both set, meaning the priority is "very high". Bits 10 and 11 represent the data size of each DMA transfer, and only bit 10 is set, representing "16 bits". Bits 8 and 9 define the data size of each DMA transfer to

the identified peripheral. Because only bit 8 is set, the size is 16 bits. Bits 5 and 7 are set, meaning that the memory increment mode and the circular buffer mode are enabled. Bit 4 represents the data transfer direction. Because it is set, the direction of data transfer is read from memory (source) out to the peripheral (destination). Bits 1 and 2 are set, signifying that the half transfer interrupt and transfer complete interrupt are enabled.

Below is the code to synchronize the DAC transfers with the drive's internal clock using a timer.

```
// DMA channel 2 updates the timer's reload register
value.
// This allows fine control of the sine wave period
dma->channel[1].config = 0x00000000;
dma->channel[1].count = TBL_LEN;
dma->channel[1].pAddr = (uint32_t)&tmr->reload;
dma->channel[1].mAddr = (uint32_t)clkTbl;
dma->channel[1].config = 0x000035B0;
```

The peripheral address of DMA channel 2 is pointing to the reload value of timer 3. The values written to the reload value of the timer are generated and stored in a lookup table "clkTbl". They will synchronize the timer with the internal clock of the drive so that the sampling rate of the drive matches the rate of transfer to produce the sine wave. One important note is that if interrupts were used to generate the sine wave, the system would experience variable latency and jitter, which would corrupt the sine wave, yielding inaccurate feedback. A timer is used to provide consistent sine wave generation in synch with the drives internal clock.

C. Implementation: ADC

Below is the code to initialize DMA channel 3 to read the ADC data.

```
dma->intClr = 0x000000FF; // only using two channels so
FF is used
dma->channel[3].config = 0x00000000;
dma->channel[3].count = 2;  // 2 channels
dma->channel[3].pAddr = (uint32_t)&adc->data; //
reading from this register in the A/D converter
dma->channel[3].mAddr = (uint32_t)adcData;  //
destination buffer
dma->channel[3].config = 0x00000583;
```

The number of data to transfer is set to 2 because the ADC must be read twice to read the secondary sine and cosine waveforms. The peripheral address of DMA channel 3 is pointing to the data register in the ADC, and the memory address is set to the beginning of the destination buffer "adcData". If the data register is not read at a consistent rate, it will fill up and be unable to store new data.

D. Oscilloscope Trace

Figure 7 below displays an oscilloscope trace of the waveforms in the system. Channel 1 displays the sinusoidal excitation signal generated from the lookup table using DMA. Channels 2 and 3 represent the cosine and sine feedback signals respectively and are fed back to the CPU via DMA. Because all three signals use DMA, the CPU is left alone to calculate the position and velocity of the motor shaft as well as perform other critical calculations or programs in real time.

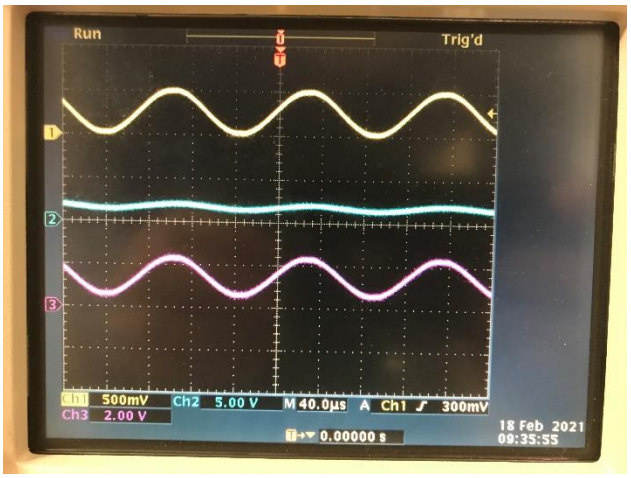

Fig. 6. Oscilloscope Trace of Resolver Waveforms

XIII. Conclusion

It is evident that DMA is essential to achieve reliable data transfer in real-time applications. The CPU experiences variable latency caused by interrupts. If the CPU were responsible for data transfer, it would not be able to maintain a real-time transfer rate as the variable latency would introduce jitter into the transfer. In the motion control industry, timing is critical. High-speed communication protocols are implemented to control large networks of down-stream devices. The Copley EtherCAT servo drive also uses DMA to transfer the EtherCAT packets in real-time. Using DMA to transfer packets on an EtherCAT network is essential to avoid CPU jitter and to keep up with update rates as low as 62.5 µsec.

Even in non-real-time applications, DMA is desired as it will reduce power consumption and improve the user experience. The IBM PC 5150 released in 1981 used DMA to handle floppy-disk I/O data transfer [5]. There was only one DMAC controlling 4 channels. Since then, the driving force behind DMA innovation has been for faster data transfers related to high-speed communications, enhanced memory storage, and high-resolution graphics.

REFERENCES

[1] Gupta, Sachin. “Optimizing Embedded Applications Using DMA.” Cypress Semiconductor Corporation, EE Times Design, Nov. 2010, www.cypress.com/file/103546/download.

[2] Chichak, Andrei. “DMA - A Little Help From My Friends.” Embedded.fm, 22 Feb. 2017, https://embedded.fm/blog/2017/2/20/an-introduction-to-dma.

[3] Catsoulis, John. Designing Embedded Hardware. O'Reilly, 2003.

[4] K., Srikanth. “RTOS Based Priority Dynamic Scheduler for Power Applications through DMA Peripherals.” Research India Publications, Sreenidhi Institute of Science and Technology, M.Tech (VLSI&ES), Department of ECE, JNTU-Hyderabad, ECIL, Hyderabad, A.P, India., Nov. 2013, www.ripublication.com/aeee/003_pp%20%20%20%20661-668.pdf.

[5] Harvey, A.F. “DMA Fundamentals on Various PC Platforms.” National Instruments Corporation, Apr. 1991, www.cs.umb.edu/~bobw/CS341/DMAFundamentals.pdf.

[6] Rubini, Alessandro, and Jonathan Corbet. Linux Device Drivers. O'Reilly, 2005.

[7] Silberschatz, Abraham, et al. Operating System Concepts. John Wiley and Sons, 2011.

[8] “DSP Direct Memory Access (DMA) Controller User's Guide (Rev. A).” Ping-Pong DMA Mode, Texas Instruments Incorporated, Mar. 2012, www.ti.com/lit/ug/spruft2a/spruft2a.pdf?ts=1612577186247&ref_url=https%253A%252F%252Fwww.google.com%252F.

[9] Franco, John. “What Is Direct Memory Access (DMA) and Why Should We Know About It?” Electrical Engineering and Computing Systems University of Cincinnati, gauss.ececs.uc.edu/Courses/c4029/lectures/dma.pdf.

[10] Castille, Kyle, and Nat Seshan. “Enhanced DMA Efficiently Manages Multiple Real-Time Data Streams.” Signal Processing Design, 13 Nov. 2004, signal-processing.mil-embedded.com/article-id/?28=.

[11] “Using Direct Memory Access (DMA) in STM32 Projects.” Embedds, Embedded Projects from around the Web, 2012, embedds.com/using-direct-memory-access-dma-in-stm23-projects/.

[12] LeCroy, Teledyne. “Resolver Output Shaft Angle.” Motion Control Tips, 29 Mar. 2019, www.motioncontroltips.com/what-features-make-resolvers-suitable-for-harsh-environments/.

Fig. 7. Copley Serial Resolver Board (FPGA Version)

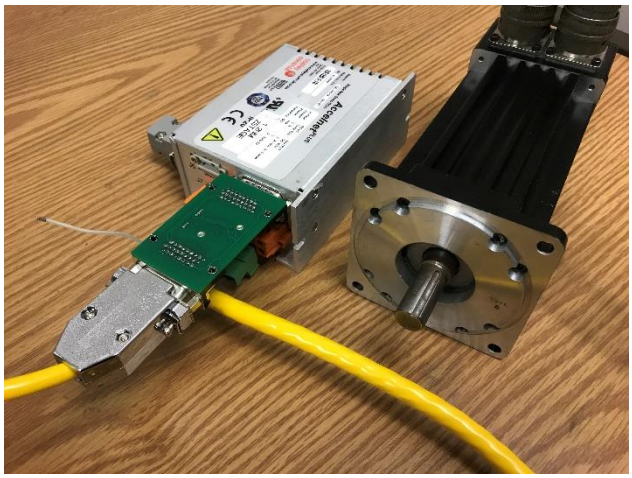

Fig. 8. Copley Serial Resolver with Drive and Motor